Capítulo

6

TCP/IP em Aplicações de Tempo Real – Um estudo de caso

Autor: Constantino Seixas Filho
UFMG – Departamento de Engenharia Eletrônica

Estudo de Caso

Introdução

Esta aula é baseada na referência [Dandass 2001] que apresenta uma aplicação de transmissão de um sinal de áudio sobre uma rede TCP/IP ou pela Internet em tempo real. O sinal de áudio é digitalizado, transmitido e tocado de volta em uma cadeia contínua, permitindo aos usuários estabelecer uma conversação em tempo real.

Cada tarefa em um sistema de tempo real tem uma data limite associada com ela e deve completar obedecendo a esta restrição de tempo. Um retardo no término teria conseqüências indesejáveis. O sinal enviado deve chegar à máquina destino exatamente a tempo de ser executado, em um tipo de entrega *just in time*.

A Internet não oferece garantia de entrega dentro de prazos de tempo estipulados. O sistema operacional Windows NT, como já estudado, também não garante que as threads serão escalonadas de forma a produzir resultados dentro de restrições de tempo adequadas.

Os atrasos na transmissão dos dados de áudio resultam em espaços vazios durante o processo de transmissão que prejudicam a qualidade da reprodução. O objetivo é diminuir a variabilidade dos atrasos entre pacotes de dados (*jitter*).

O *jitter* é definido como a variação no tempo de latência dos pacotes de dados (tempo de trajeto da origem até o destino final).

Funcionamento:

Mecanismo de transmissão:

1) A aplicação fornece buffers de dados para o dispositivo de captura de dados (waveIn).
2) O dispositivo enche cada buffer com dados digitalizados e os retorna para a aplicação.
3) A aplicação insere o buffer cheio na fila de transmissão e após o dado ser colocado na rede a aplicação retorna o buffer vazio para o dispositivo waveIn.

Mecanismo de recepção:

1. Na reprodução, pacotes de dados de áudio são recebidos da rede em buffers retirados de uma lista de buffers livres.
2. A aplicação insere os buffers cheios na lista de reprodução. A lista de reprodução é usada para montar os buffers na ordem adequada para a reprodução. Após a ordenação dos buffers, a aplicação entrega os buffers para o dispositivo de reprodução waveOut.

3. Após tocar os dados no buffer, waveOut retorna os buffers para a aplicação para reinserção na lista de buffers livres.

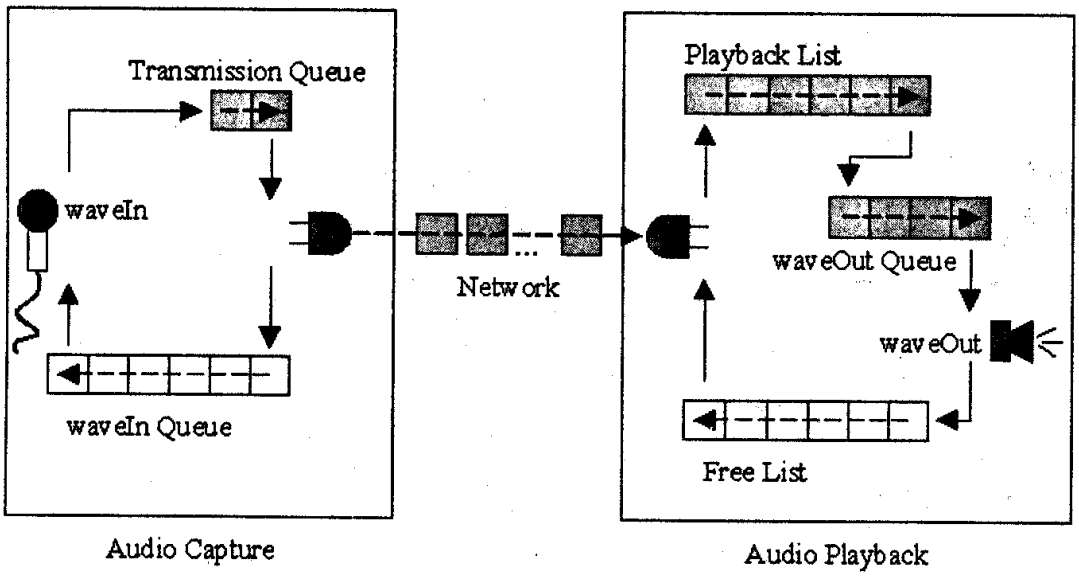


Figura 1 - Esquema do programa mysound

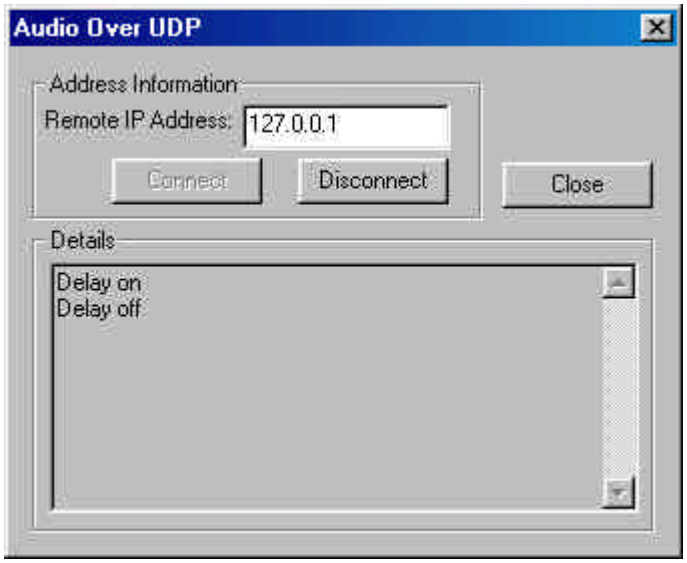


Figura 2 - Janela do programa Audio Over UDP – operação em um único micro

TCP vs UDP

Como nesta aplicação a entrega dos dados em tempo é mais importante que a entrega dos dados sem erros, o protocolo UDP foi preferido em relação ao TCP.
Pacotes TCP são entregues em ordem e livres de erros. O protocolo retransmite os pacotes defeituosos ou faltantes automaticamente. Isto provoca atrasos que dificulta o controle de *jitter*.
O protocolo UDP não garante nem a ordem, nem a correção nem a integridade dos pacotes. Pacotes podem chegar corrompidos ou simplesmente não chegar. Supondo que apenas um pequeno percentual de pacotes cheguem danificados, o protocolo UDP/Ip foi o escolhido.

Para evitar os erros decorrentes da perda de pacotes, os dados do pacote anterior são repetidos em cada novo pacote enviado. Desta forma mesmo que um pacote em cada dois seja perdido, a reprodução dos dados será perfeita. Este mecanismo entretanto irá implicar na necessidade de maior banda de passagem.

Os programas:
mysound.cpp é o programa que proporciona a troca de mensagens de áudio entre dois computadores. Para teste com um único computador, um segundo programa (soundeco.cpp) é usado como um servidor de eco que reenvia os pacotes para serem reproduzidos. O servidor de eco elimina propositadamente alguns pacotes para que o comportamento do algoritmo de recuperação possa ser observado.

Aquisição de áudio

As velocidades típicas de aquisição de áudio são:

Velocidade de amostragem (kHz)	Qualidade	Resolução bits/amostra	Banda de passagem (bytes/s)
8.0	Telefone	8	8 000
11.02	Telefone	8	11 020
22.05	Rádio	8	22 050
44.1	CD	16	176 400 (stereo)

Devido à grande banda de passagem exigida, um mecanismo de compressão de dados deve ser utilizado.
Deve-se estudar qual drive codec (***co****mpression/****dec****ompression*) utilizar, dependendo da qualidade de áudio pretendida e das características de compressão objetivadas.

Nesta aplicação será utilizado o codec GSM 6.10. Este codec amostra dados a 8 kHz e utiliza um algoritmo de compressão com perdas, isto é, os sinais comprimidos e descomprimidos não serão idênticos.

O GSM garante uma saída com apenas 1 625 bytes/s, bem adequada para uso em canais de baixa capacidade. Outros codecs como PCM, por exemplo, digitalizam

em mono ou estéreo com resolução de 8 ou 16 bits, sem nenhuma compressão de dados.

Dados do codec	
Sample rate	8 kHz
Num channels	1 (mono)
Data rate	1625 bytes/s
Block alignment	65
Samples per block	320
Bits per sample	0 (determinado pelo GSM)

Serão produzidos blocos de dados de 65 bytes, 25 vezes por segundo (1625 bytes/s / 65 bytes) e cada bloco conterá (8000/25) 320 amostras. Cada bloco de 65 bytes contém 40 ms (1000 ms / 25) de dados de áudio.

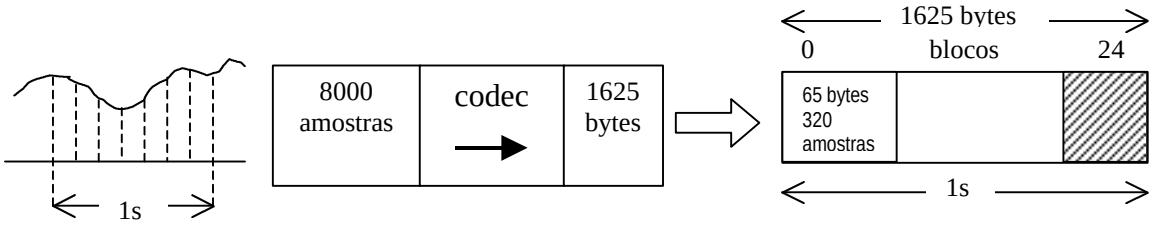


Figura 3 - amostragem e compressão de dados com o codec GSM 6.10

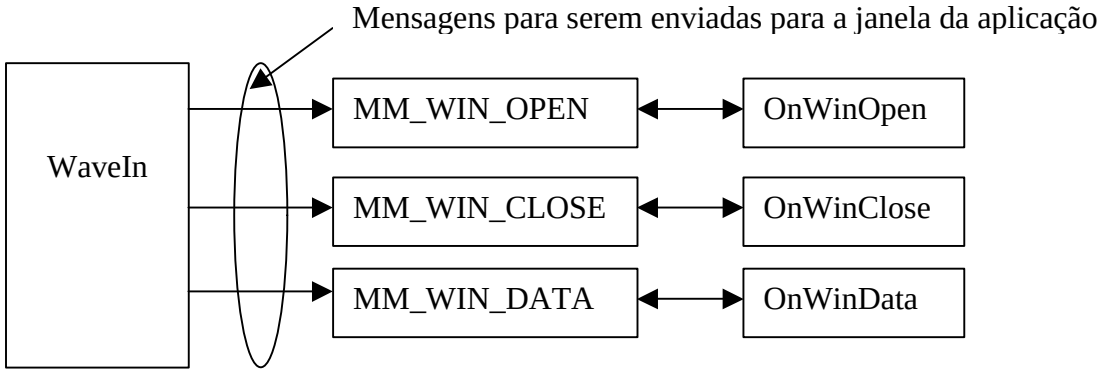


Figura 4 - Mensagens enviadas por WaveIn para a janela da aplicação

Você abre um dispositivo WaveIn e requisita um codec e um conjunto de características de áudio (taxa de amostragem, resolução, canais, etc.) passando uma estrutura WAVEFORMATEX, para a função WaveInOpen(),.
A rotina WaveInOpen também especifica que o dispositivo WaveIn deve enviar mensagens para a janela da aplicação para reportar informações de status e para retornar buffers cheios para a aplicação. Como exercício você poderia utilizar uma função *callback*, ou uma outra thread para o mesmo propósito. As mensagens enviadas para a janela e as rotinas de tratamento destes eventos são mostradas na Figura 4.

A inicialização do codec é feita no método OnConnect da classe CsoundDialog.

WaveInOpen retorna um handle (em phwi) para o dispositivo de entrada a ser usado nas futuras operações.

w a v e I n O p e n

MMRESULT waveInOpen(

```
LPHWAVEIN phwi,              // Apontador para handle do dispositivo
UINT uDeviceID,              // Identificador do dispositivo
LPWAVEFORMATEX pwfx,         // Formato desejado para gravação
DWORD dwCallback,            // Função callback, janela, evento, ...
DWORD dwCallbackInstance,    // Parâmetro para função callback
DWORD fdwOpen                // Atributos e flags
);
```

Comentários sobre os parâmetros:

phwi	Apontador para um buffer que receberá um handle identificando o dispositivo de áudio aberto. O handle será usado para identificar o dispositivo, ao usar as demais funções de áudio.
uDeviceID	Identificador do dispositivo a ser aberto. Pode ser um identificador de dispositivo ou um handle para um dispositivo de entrada de áudio já aberto. O flag WAVE_MAPPER seleciona um dispositivo de entrada de áudio capaz de gravar no formato especificado.
pwfx	Apontador para uma estrutura WAVEFORMATEX que identifica o formato desejado para gravação de dados em **formato wave**. Esta estrutura pode ser liberada assim que a função waveInOpen retornar.

```
typedef struct {
WORD        wFormatTag;
WORD        nChannels;
DWORD       nSamplesPerSec;
DWORD       nAvgBytesPerSec;
WORD        nBlockAlign;
WORD        wBitsPerSample;
WORD        cbSize;
} WAVEFORMATEX;
```

dwCallback	Apontador para uma função *callback*, um handle para um evento ou para uma janela ou o identificador de uma thread a ser chamada durante a gravação de um sinal de áudio para processar as mensagens relacionadas com o processo de gravação. Se nenhuma função for requerida, este valor será 0.
dwCallbackInstance	Dado a ser passado para a rotina de *callback*. Não é usado caso uma janela é que for designada para tratar o evento.

fdwOpen	Flags de abertura do dispositivo: • CALLBACK_EVENT O parâmetro *dwCallback* é um handle para evento. • CALLBACK_FUNCTION O parâmetro *dwCallback* é um endereço para um procedimento de callback • CALLBACK_NULL Nenhum mecanismo de *callback* é utilizado. Este é o valor default. • CALLBACK_THREAD O parâmetro *dwCallback* é o identificador de uma thread • CALLBACK_WINDOW O parâmetro *dwCallback* é o handle para uma janela • WAVE_FORMAT_DIRECT Se esta flag for definida o driver ACM não realiza conversões no dado de áudio • WAVE_FORMAT_QUERY A função indaga ao dispositivo para determinar se ele suporta o formato dado, mas não abre o dispositivo. • WAVE_MAPPED O parâmetro *uDeviceID* especifica o dispositivo de áudio a ser mapeado pelo mapeador de wave.

Retorno da função:

Status	Interpretação
MMSYSERR_NOERROR	Sucesso
MMSYSERR_ALLOCATED	O recurso especificado já está alocado
MMSYSERR_BADDEVICEID	Identificador de dispositivo está fora de faixa
MMSYSERR_NODRIVER	Nenhum device driver está presente
MMSYSERR_NOMEM	Não foi possível alocar ou reservar memória
WAVERR_BADFORMAT	Tentativa de abrir com um formato de áudio wave não suportado

Observações:

- Use a função waveInGetNumDevs para determinar o número de dispositivos de áudio presentes no sistema. O identificador de dispositivo especificado por uDeviceID varia de 0 a número máximo de dispositivos –1.

- Se você escolher que uma thread ou janela receberá as informações de callback, as seguintes mensagens serão enviadas para o procedimento da

janela ou thread para indicar o progresso da entrada de áudio: MM_WIM_OPEN, MM_WIM_CLOSE, e MM_WIM_DATA.

Transmissão de dados

Em OnConnect o socket UDP é criado e a função connect é chamada para estabelecer um endereço remoto para comunicação via sockets. É utilizada a porta número 1500.

Rotina de inicialização

```
class CSoundDialog {
protected:
      HWND        m_hWnd;       // Handle para janela de diálogo
      HWAVEIN     m_hWaveIn;    // Handle para dispositivo de captura de áudio
      CsendBuffer  m_aInBlocks[NUM_BLOCKS]; // Buffers de captura
      int m_iCountIn;           // Itens na fila de captura
      DWORD       m_dwOutSeq;  // Contador de seqüência de blocos enviados
      ****

void OnConnect() {     // método da classe CSoundDialog
    char              szIPAddress[128];
    unsigned long     ulAddrIP;
    struct hostent    *pHostEnt;
    GSM610WAVEFORMAT     WaveFormatGSM;
    MMRESULT          mmRC;

    ZeroMemory(&m_SockAddr, sizeof(m_SockAddr));
    m_nPrevSize = 0;  // Inicializa tamanho do buffer anterior

    // Obtém endereço IP remoto do host
    GetDlgItemText(m_hWnd,IDC_EDIT_REMOTEIPADDR,szIPAddress,
    sizeof(szIPAddress));
    ulAddrIP = inet_addr(szIPAddress);
    if (ulAddrIP != INADDR_NONE)    // Endereço na forma x.y.z.w ?
       memcpy(&(m_SockAddr.sin_addr), &ulAddrIP, sizeof(m_SockAddr.sin_addr));
    else { // Use DNS para obter endereço IP
       pHostEnt = gethostbyname(szIPAddress);
       if (pHostEnt == NULL) {
            MessageBox(m_hWnd, "Erro resolvendo nome remoto", "Erro",
            MB_OK | MB_ICONSTOP);
            return;
       }
       memcpy(&(m_SockAddr.sin_addr), pHostEnt->h_addr, pHostEnt->
       h_length);
    }

    // Cria um socket e o associa a um port
    m_Socket = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);
    m_SockAddr.sin_family = AF_INET;
```

```
m_SockAddr.sin_port = htons(PORT_NUMBER);
bind(m_Socket, (sockaddr*)&m_SockAddr, sizeof(m_SockAddr));
// Define o endereço remoto m_SockAddr para comunicações futuras
// Conecta
connect(m_Socket, (struct sockaddr*)&m_SockAddr, sizeof(m_SockAddr));

// Inicializa o dispositivo de entrada de dados: wave input
// Abre dispositivo de captura e reprodução para GSM 6.10
WaveFormatGSM.wfx.wFormatTag = WAVE_FORMAT_GSM610;
WaveFormatGSM.wfx.nChannels = 1;              // mono
WaveFormatGSM.wfx.nSamplesPerSec = 8000; // sample rate
WaveFormatGSM.wfx.nAvgBytesPerSec = 1625; // data rate = 1625 bytes/s
// Block alignment = menor quant de dados codec pode processar de uma vez
WaveFormatGSM.wfx.nBlockAlign = 65;
WaveFormatGSM.wfx.wBitsPerSample = 0;
// para este codec o número de bits por amostra não é especificado
WaveFormatGSM.wfx.cbSize = 2;
// bytes de info extra apendados ao final da estrutura WAVEFORMATEX
WaveFormatGSM.wSamplesPerBlock = 320;

// Abre dispositivo de reprodução
mmRC = waveOutOpen(&m_hWaveOut,  // handle do dispositivo
                        (UINT)WAVE_MAPPER, // Id do dispositivo
                        (LPWAVEFORMATEX)&(WaveFormatGSM.wfx),
                        (DWORD)m_hWnd, // função callback, janela, ev...
                        (DWORD)NULL, // parâmetro função callback
                        CALLBACK_WINDOW); // callback é handle jan.

if (mmRC != MMSYSERR_NOERROR)
     Report("Erro abrindo dispositivo de reprodução wave\r\n");
else
     m_fOutClosing = false;

// Abre dispositivo de entrada
mmRC = waveInOpen(&m_hWaveIn,
                        (UINT)WAVE_MAPPER,
                        (LPWAVEFORMATEX)&(WaveFormatGSM.wfx),
                        (DWORD)m_hWnd,
                        (DWORD)NULL,
                        CALLBACK_WINDOW);

if (mmRC != MMSYSERR_NOERROR)
     Report("Não conseguiu abrir dispositivo de entrada wave\r\n");
else {
     m_fInClosing = false;
     waveInStart(m_hWaveIn);
}

if (!(m_fInClosing && m_fOutClosing)) {
   // Se pelo menos um dos dispositivos foi iniciado
```

```
            EnableWindow(GetDlgItem(m_hWnd, IDC_BUTTON_CONNECT),
            FALSE);  // Desabilita botão connect
            EnableWindow(GetDlgItem(m_hWnd, IDC_BUTTON_DISCONNECT),
            TRUE);    // Habilita botão disconnect
        } // if
}
```

Tratamento das Mensagens:

MM_WIM_OPEN

Avisa que o dispositivo waveIn foi aberto. OnWimOpen() será chamada. OnWimOpen prepara 25 bufers de entrada, cada qual correspondendo a um bloco de 65 bytes de dados de áudio.

O bloco de dados a ser transmitido é definido pela estrutura XMITDATA.

NumSeq	Tamanho Bloco	Tamanho Bloco P (previous)	Bloco	BlocoP (previous)

O número de seqüência será utilizado pelo computador que recebe os blocos, para tocar a mensagem na ordem apropriada.

```
typedef struct {
        DWORD   m_dwSeq;  // Número de seqüência do bloco
        T_BSIZE   m_nSize;    // Tamanho do bloco em bytes
        T_BSIZE   m_nSizeP;  // Tamanho do bloco anterior (previous)
        BYTE      m_abData[BLOCK_SIZE];   // Bloco de dados a ser transmitido
        BYTE      m_abDataP[BLOCK_SIZE]; // Bloco de dados anterior
} XMITDATA;
```

Cada bloco de dados será precedido por um cabeçalho representado pela estrutura WAVEHDR:

```
/* cabeçalho do bloco wave */
typedef struct wavehdr_tag {
        LPSTR lpData;                 /* apontador para buffer de dados */
        DWORD dwBufferLength;         /* tamanho do bloco em bytes */
        DWORD dwBytesRecorded;        /* usado apenas para entrada */
        DWORD dwUser;                 /* livre para uso do cliente */
        DWORD dwFlags;                /* flags (veja definições) */
        DWORD dwLoops;                /* contador de loops de controle */
        struct wavehdr_tag FAR *lpNext; /* reservado para driver */
        DWORD reserved;               /* reservado para driver */
} WAVEHDR, *PWAVEHDR, NEAR *NPWAVEHDR, FAR *LPWAVEHDR;
```

// O bloco de dados completo é representado pela classe CSendBuffer

```
// Prepara 25 buffers e envia para dispositivo waveIn encher
void OnWimOpen() {
        m_dwOutSeq = 0;        // reseta seqüência de blocos enviados
        m_iCountIn = 0;        // reseta contador de blocos na fila
        for (int i = 0; i < NUM_BLOCKS; i++) { // são 25 buffers
              // prepara e adiciona blocos para capturar a fila do dispositivo
              m_aInBlocks[i].Prepare(m_hWaveIn);  // Prepara buffer
              m_aInBlocks[i].Add(m_hWaveIn);      // Envia para dispositivo encher
              m_iCountIn++;
        }
}
```

CSendBuffer
WAVEHDR m_WaveHeader; XMITDATA m_Data;
Prepare(); Unprepare(); Add();

```
class CSendBuffer {
public:
    WAVEHDR   m_WaveHeader; // cabeçalho do buffer
    XMITDATA  m_Data;            // Bloco de dados a ser transmitido via UDP

    MMRESULT Prepare(HWAVEIN hWaveIn) { // Prepara para reproduzir
        ZeroMemory(&m_WaveHeader, sizeof(m_WaveHeader));
        m_WaveHeader.dwBufferLength = BLOCK_SIZE;          // 25
        m_WaveHeader.lpData = (char*)(m_Data.m_abData);      // Dados
        m_WaveHeader.dwUser = (DWORD)this; // Aponta objeto CsendBuffer
        // Campo livre para uso do usuário. Não é usado pelas funções de áudio
        return waveInPrepareHeader(hWaveIn, &m_WaveHeader,
                sizeof(m_WaveHeader)); // prepara buffer para entrada de áudio
    }  // Prepare

    MMRESULT Unprepare(HWAVEIN hWaveIn) {
        // Deve ser chamada depois que o device driver enche um buffer
        // Desconecta buffer do dispositivo de entrada
        return  waveInUnprepareHeader(hWaveIn,m_WaveHeader,
                sizeof(m_WaveHeader));
    }  // Unprepare

    MMRESULT Add(HWAVEIN hWaveIn) {
        // Adiciona buffer à fila do dispositivo
        return waveInAddBuffer(hWaveIn, &m_WaveHeader, sizeof(m_WaveHeader));
    }  // Add
};
```

M M _ W I M _ D A T A

O dispositivo waveIn irá retornar buffers cheios para a aplicação.
O membro dwBytesRecorded de WAVEHDR contém o número de bytes colocados no buffer pela aplicação. OnWimData irá retirar os dados.

```
// Define template fila de buffers de envio
typedef std::queue<CSendBuffer*>  CSendBufQ;

class CSoundDialog {
protected:
   ***
   T_BSIZE       m_nPrevSize;     // Tamanho do bloco de dados anterior
   CSendBufQ     m_qpXmitBufs;  // Fila de transmissão
   ***

void OnWimData(WAVEHDR *pHdrWave) { // retorna pointer para cabeçalho
     CsendBuffer    *pAudioBuffer;        // apontador para o wave buffer
     XMITDATA    *pXmitData;           // ptr para parte a ser enviada

     m_iCountIn--;   // Decrementa contador de buffers entregues para WaveIn
     pAudioBuffer = (CSendBuffer*)(pHdrWave->dwUser);
     // aponta para o próprio CSendBuffer
     // Desconecta buffer do dispositivo de captura
     pAudioBuffer->Unprepare(m_hWaveIn);
     if (!m_fInClosing) {
          // captura de áudio terminando ? Neste caso, os buffers vazios retornam à
          // aplicação através de mensagens MM_WIM_DATA.
          pXmitData = &(pAudioBuffer->m_Data); // aponta campo de dados

          // Define buffer: tamanho do dado, seqüência, dados redundantes
          pXmitData->m_nSize = (T_BSIZE)(pHdrWave->dwBytesRecorded);
          pXmitData->m_dwSeq = m_dwOutSeq++;
          pXmitData->m_nSizeP = m_nPrevSize;
          // m_abPrevData guarda uma cópia dos dados anteriores
          // m_nPrevSize guarda quantos bytes havia na mensagem anterior
          memcpy(pXmitData->m_abDataP, m_abPrevData, m_nPrevSize);

          // Salva uma cópia do dado a ser enviado com o próximo pacote
          m_nPrevSize = pXmitData->m_nSize;
          memcpy(m_abPrevData, pXmitData->m_abData, m_nPrevSize);

          // insere novo buffer na fila de transmissão
          m_qpXmitBufs.push(pAudioBuffer);
          OnSocketWrite();    // Tente enviar buffers enfileirados
     }
     else { // pedido de fechamento não volte a executar
          // Se todos os buffers tiverem retornado, feche o dispositivo
          if (m_iCountIn == 0)
                waveInClose(m_hWaveIn);
            }
     } // OnWimData
```

OnWimData faz uma chamada a OnSocketWrite() para enviar dados.

```
// Define template fila de buffers de envio
typedef std::queue<CSendBuffer*>        CSendBufQ;

class CSoundDialog {
protected:
   ***
   CSendBufQ m_qpXmitBufs; // Fila de transmissão
   ***

// Procedimento para envio de todos os quadros da fila de transmissão via UDP
void OnSocketWrite() {
     CSendBuffer *pBuffer;

     if (m_fInClosing) return;              // Não transmite se estiver fechando

     while (!m_qpXmitBufs.empty()) {    // Enquanto fila não vazia
        pBuffer = m_qpXmitBufs.front();// Pega cabeça da fila
        // Envie dados pelo soquete
        if (send(m_Socket, (char*)&(pBuffer->m_Data), sizeof(XMITDATA), 0)
            == SOCKET_ERROR) {
        Report("Erro enviando dados\r\n");
        break;                  // Pare quando blocos UDP encherem
     } // while
     // Remover e reciclar o buffer de envio
     m_qpXmitBufs.pop();                 // Remove nodo da fila
     pBuffer->Prepare(m_hWaveIn);       // Prepara
     pBuffer->Add(m_hWaveIn);           // Adiciona à fila de buffers livres
     m_iCountIn++;
     }
}// OnSocketWrite
```

M M _ W I M _ C L O S E

Dispositivo waveIn foi fechado.

```
void OnWimClose() {
     m_hWaveIn = 0;
     if (m_hWaveOut == 0) {         // Se ambos os dispositivos estão fechados
        EnableWindow(GetDlgItem(m_hWnd, IDC_BUTTON_DISCONNECT),
                                    FALSE);    // Desabilita botão desconexão
        EnableWindow(GetDlgItem(m_hWnd, IDC_BUTTON_CONNECT),
                                    TRUE);     // Habilita botão de conexão
        if (m_fExiting)
            EndDialog(m_hWnd, 0);
     }
}      // OnWimClose
```

É invocado quando o dispositivo de reprodução é aberto.

```
// Define template lista de buffers de recepção
typedef std::list<CRecvBuffer*>   CRecvBufL;

class CSoundDialog {
protected:
    ***
    HWAVEOUT        m_hWaveOut; // Handle para dispositivo de reprodução
    // Buffers de reprodução
    CrecvBuffer     m_aOutBlocks[NUM_BLOCKS*2];
    CrecvBufL       m_lpFreeBufs;  // Lista de buffers de recepção vazios
    ***

void OnWomOpen() {
     m_iCountOut = 0;        // reseta contador de blocos na fila de WaveOut
     m_dwSeqExp = 0;         // reseta contador de blocos recebidos

     for (int i = 0; i < NUM_BLOCKS*2; i++) {   // são 2 * 25 buffers
        m_aOutBlocks[i].Prepare(m_hWaveOut);
        m_lpFreeBufs.push_back(&(m_aOutBlocks[i]));  // insere no final da lista
     } // for
     WSAAsyncSelect(
               m_Socket,               // Soquete para espera de evento
               m_hWnd,                 // Janela a ser notificada
               WM_USR_SOCKIO,          // Mensagem a ser enviada p/ janela
               FD_READ | FD_WRITE // Evento: Pronto para ler ou escrever
     );        // socket não bloqueante
}      // OnWomOpen
```

A principal rotina utilizada é WSAAsyncSelect. Esta rotina coloca o socket em modo assíncrono, isto é, não bloqueante. Toda vez que um datagrama puder ser lido ou recebido pelo socket, uma mensagem será enviada para a janela (WM_USR_SOCKIO). Desta forma a aplicação não ficará bloqueada e continuará a processar mensagens relativas à captura de áudio, processamento da interface com o usuário, etc.

CRecvBuffer
WAVEHDR m_WaveHeader; XMITDATA m_Data;
Prepare(); Unprepare(); Add();

```
class CRecvBuffer {
public:
     WAVEHDR        m_WaveHeader; // cabeçalho do buffer
     XMITDATA       m_Data;     // Bloco de dados a ser transmitido via UDP
```

```
MMRESULT Prepare(HWAVEOUT hWaveOut) {
    ZeroMemory(&m_WaveHeader, sizeof(m_WaveHeader));
    m_WaveHeader.dwBufferLength = BLOCK_SIZE;
    m_WaveHeader.lpData = (char*)(m_Data.m_abData);
    // Campo livre para uso do usuário. Não é usado pelas funções de áudio
    m_WaveHeader.dwUser = (DWORD)this; // aponta obj. CrecvBuffer
    // Prepara um bloco de áudio para playback
    return waveOutPrepareHeader(hWaveOut, &m_WaveHeader,
                                sizeof(m_WaveHeader));
}  // Prepare

MMRESULT Unprepare(HWAVEOUT hWaveOut) {
    // Deve ser chamada depois que o device driver reproduziu o bloco de dados
    // Você deve chamar esta função antes de liberar o buffer
    return waveOutUnprepareHeader(hWaveOut, &m_WaveHeader,
                                  sizeof(m_WaveHeader));
}  // Unprepare

MMRESULT Add(HWAVEOUT hWaveOut) {
    // Envia um bloco de dados ao dispositivo de reprodução de áudio
    return waveOutWrite(hWaveOut, &m_WaveHeader,
                        sizeof(m_WaveHeader));
  }  // Add
};
```

Recepção de dados

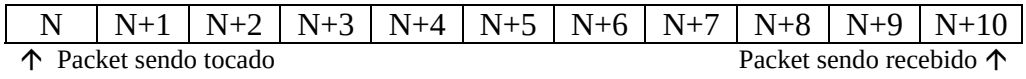
Toda vez que um dado for recebido, o socket gerará a mensagem WM_USR_SOCKIO. Caso o evento seja a recepção de um packet, OnSocketRead será chamado.

```
***
case WM_USR_SOCKIO:
      if (WSAGETSELECTEVENT(lParam) == FD_READ)
            pSoundDlg->OnSocketRead();
      if (WSAGETSELECTEVENT(lParam) == FD_WRITE)
            pSoundDlg->OnSocketWrite();
      break;
****

typedef CRecvBufL::iterator      CBufLIter;

class CSoundDialog {
protected:
    ***
    boolm_fOutClosing;          // Interrompendo reprodução ?
    SOCKETm_Socket;             // UDP socket
    DWORD m_dwSeqExp;           // Seqüência do bloco de dados esperado na
                                // cabeça da lista de reprodução
```

```
    CRecvBufL m_lpPlayBufs;     // Lista de buffers de reprodução
                                // em ordem ascendente
    CRecvBufL m_lpFreeBufs;     // Lista de buffers de recepção vazios
    ***

void OnSocketRead() {
    CrecvBuffer         *pBuffer;
    XMITDATA        *pData;

    if (m_fOutClosing) return;  // Ignora dados se reprodução está encerrando

    if (m_lpFreeBufs.empty()) { // Fila de buffers livres vazia
        XMITDATA   Data;  // Usa buffer temporário para receber dado

        // Lê dados do port e descarata
        recv(m_Socket, (char*)&Data, sizeof(Data), 0);
        Report("Não existem buffers livres (descartando bloco)\r\n?");
        return;
    }
    pBuffer = (CRecvBuffer*)(m_lpFreeBufs.front()); // Aponta buf livre na fila
    pData = &(pBuffer->m_Data);
    if (recv(m_Socket, (char*)pData, sizeof(*pData), 0) == SOCKET_ERROR)
        Report("Erro recebendo dados\r\n");
    else {
        if (pData->m_dwSeq == 0)        // Primeiro bloco da seqüência chegou
            m_dwSeqExp = 0;             // Reseta a seqüência esperada

        if (pData->m_dwSeq >= m_dwSeqExp) {
            CBufLIter Iter;

            // Procura a posição apropriada
            for (Iter = m_lpPlayBufs.begin(); Iter != m_lpPlayBufs.end(); Iter++)
              if ((*Iter)->m_Data.m_dwSeq == pData->m_dwSeq)
                  return; // Buffer duplicado: não insere
              else if ((*Iter)->m_Data.m_dwSeq >  pData->m_dwSeq)
                  break;  // Ponto de inserção encontrado !

            m_lpFreeBufs.pop_front();           // Remove da lista de buffers livres
            m_lpPlayBufs.insert(Iter, pBuffer);// Insere na lista de buffers  reprod.

            JitterControl();
        }  // if
    }// else
}  // OnSocketRead
```

OnSocketRead() toma um buffer livre (instância da classe CrecvBuffer) da lista de buffers livres (m_lpFreeBufs), recebe os dados do socket no buffer, e insere o buffer cheio na fila de buffers a serem executados (m_lpPayBufs). Existem 50 instâncias da classe CRecvBuffer disponíveis na aplicação.

Como os packets podem chegar com muito atraso é dada uma margem de segurança de 10 packets de atraso entre a recepção e a reprodução, isto é, o packet n será tocado quando o packet n + 10 estiver sendo recebido.
Se um packet não chegar 200 ms do tempo em que se tornou necessário para reprodução, ele é assumido como perdido e o próximo buffer é preparado para tocar.



JitterControl() é chamado quando um datagrama é recebido e quando um buffer é retornado para a aplicação após ser reproduzido.

Se o número de seqüência do bloco na cabeça da lista de reprodução é o número de seqüência esperado, JitterControl() remove o buffer da lista de reprodução e o entrega a waveOut para tocar.

Se o bloco esperado está faltando, mas o próximo está disponível, o dado do bloco esperado é recuperado através de *RecoverPrevData()* da cópia redundante do próximo bloco de dados e é colocado na lista de saída, assim como o bloco seguinte.

Se o número do bloco na cabeça da lista não é o esperado e o número de blocos disponíveis é menor que 5, não podemos esperar mais e o número do bloco esperado é feito igual ao número do primeiro bloco disponível. O bloco faltante é dado como perdido.

CONTROLE DE JITTER

```
#define THRESHOLD                10    // Atraso para tocar = 400 ms
#define PLAYBACK_THRESHOLD       5     // Limite para espera de um pacote

class CSoundDialog {
protected:
    ***
    HWAVEOUT  m_hWaveOut; // Handle para dispositivo de saída
    int m_iCountOut;          // Itens na fila de reprodução
    DWORD m_dwSeqExp;         // Número seqüencial do bloco de dados
                              // esperado na cabeça da lista de reprodução
    CRecvBufL m_lpPlayBufs;   // Lista de buffers de reprodução
                              // em ordem ascendente
    bool    m_fDelay;         // Flag: define se está em modo atraso:
                              // blocos serão acumulados antes de tocar
    ***

void JitterControl() {
    CRecvBuffer *pBuffer;

    if (m_fDelay) { // Será computado um atraso de segurança para iniciar reprod.
```

```
        if (m_lpPlayBufs.size() >= THRESHOLD) { // buffers > 10 ?
            // Inicia reprodução se existe um número suficiente de buffers recebidos
            Report("Atraso desligado\r\n");
            m_fDelay = false;  // Desliga atraso para próximos pacotes

            for (int i = 0; i < THRESHOLD; i++) {
                pBuffer = m_lpPlayBufs.front();  // aponta para primeiro buffer
                m_lpPlayBufs.pop_front();        // retira buffer da fila

                // Espera bloco N, mas é o N+1 que está disponível
                if (pBuffer->m_Data.m_dwSeq == (m_dwSeqExp+1)) {
                    // Recupera do bloco anterior se buffer está faltando
                    RecoverPrevData(pBuffer);  // insere bloco N na fila de reprod.
                    i++;
                    pBuffer->Prepare(m_hWaveOut);    // Prepara buffer para reprod
                    pBuffer->Add(m_hWaveOut);        // Insere na lista de WaveOut
                } else { // toca o bloco certo ou qualquer outro
                    pBuffer->Prepare(m_hWaveOut);    //  Prepara buffer para reprod
                    pBuffer->Add(m_hWaveOut);        // Insere na lista de WaveOut
                }
                m_iCountOut++;  // incrementa número de blocos na fila para tocar
                m_dwSeqExp = pBuffer->m_Data.m_dwSeq + 1;
            }  // for
        } // if buffers > 10
    return;
    } // if atraso sendo computado

    if (m_iCountOut == 0) {
        // Inicia atraso se este é o primeiro buffer a ser recebido
        m_fDelay = true;
        Report("Atraso ligado\r\n");
        return;
    }

    for (;;) {// Reproduza quanto for possível, sem gaps
        if (m_lpPlayBufs.empty())  return;  // lista de reprodução vazia

        pBuffer = m_lpPlayBufs.front(); // aponta primeiro da fila

        // Espera bloco N, mas é o N+1 que está disponível
        if (pBuffer->m_Data.m_dwSeq == (m_dwSeqExp+1)) {
            // Recupera bloco faltante e o insere na fila para tocar
            RecoverPrevData(pBuffer);
            m_dwSeqExp++;
        }

        // Este é o buffer esperado: toque
        if (pBuffer->m_Data.m_dwSeq == m_dwSeqExp) {
            pBuffer->Prepare(m_hWaveOut);    // Prepara buffer para reprod
            pBuffer->Add(m_hWaveOut);        // Insere na lista de WaveOut
```

```
            m_iCountOut++;                  // Contador de blocos para tocar
            m_dwSeqExp = pBuffer->m_Data.m_dwSeq + 1;
            m_lpPlayBufs.pop_front();       // Retira bloco da fila
            continue;
        }

        if (m_iCountOut < PLAYBACK_THRESHOLD) { // dados para tocar < 5
            // Toque o próximo buffer independentemente do número de seq
            // Porque estamos com pouco dado
            // Número de seqüência esperado é feito igual ao próximo disponível
            m_dwSeqExp = pBuffer->m_Data.m_dwSeq;
            Report("pulando...\r\n");
            continue;
        }
        break;
    }
} // for


void RecoverPrevData(CRecvBuffer *pBuffer) {
    // Dado um quadro recebido, copia o bloco correspondente ao quadro
    // anterior em um buffer e o insere na fila de reprodução

    CRecvBuffer *pBufferP;      // Buffer para quadro anterior (previous)

    if (m_lpFreeBufs.empty()) {    // Falha se não há buffers livres
        Report("Recuperação falhou (faltam buffers livres)\r\n");
        return;
    }
    pBufferP = m_lpFreeBufs.front();    // Aponta buffer livre na cabeça da fila
    m_lpFreeBufs.pop_front();           // Retira apontador para buffer da fila

    // pBufferP contém um apontador par um buffer livre
    // copia tamanho do campo de dados e o próprio campo de dados
    pBufferP->m_Data.m_nSize = pBuffer->m_Data.m_nSizeP;
    memcpy( pBufferP->m_Data.m_abData, pBuffer->m_Data.m_abDataP,
            pBufferP->m_Data.m_nSize);

    pBufferP->Prepare(m_hWaveOut);  // Prepara buffer para reprodução
    pBufferP->Add(m_hWaveOut);      // Insere na lista de WaveOut
    m_iCountOut++;  // incrementa número de blocos na fila para tocar
}     // RecoverPrevData
```

Quando o dispositivo waveOut toca o dado no buffer, o buffer é retornado para a aplicação e a mensagem WOM_DONE é enviada. Esta mensagem é tratada por OnWomDone().

```
class CSoundDialog {
protected:
```

```
   ***
   bool              m_fOutClosing;  // Parando reprodução ?
   HWAVEOUT          m_hWaveOut;     // Handle para dispositivo de saída
   Int               m_iCountOut;    // Itens na fila de reprodução
   CrecvBufL         m_lpFreeBufs;   // Lista de buffers de recepção livres
   ***

void OnWomDone(WAVEHDR *pHdrWave) {
     CRecvBuffer *pBuffer;

     // Playback don -- Unprepare buffer and add to free list
     pBuffer = (CRecvBuffer*)(pHdrWave->dwUser);
     pBuffer->Unprepare(m_hWaveOut); // desmonta buffer
     m_iCountOut--;     // Decrementa número de blocos a serem tocados
     m_lpFreeBufs.push_back(pBuffer);   // Insere buffer na lista de buffers livres

     if (!m_fOutClosing)              // se não estiver saindo
        JitterControl();              // Realiza controle de jitter
     else if (m_iCountOut == 0)       // Fila de saída vazia
              waveOutClose(m_hWaveOut);
}  // OnWomDone
```

Terminação do Programa

Antes de sair, a aplicação deve interromper a captura de áudio e o processo de reprodução, recuperar todos os buffers dos dispositivos waveIn e waveOut e fechá-los. Se o programa sair antes dos dispositivos estarem fechados, o sistema multimídia ficará bloqueado. Os dispositivos não podem ser fechados antes que tenham devolvido todos os buffers para a aplicação. O flag m_fExiting é ativado em OnCancel, indicando que a aplicação está saindo e OnDisconnect é chamado.

```
void OnCancel() {
     if ((m_hWaveOut != 0) || (m_hWaveIn != 0)) {
        OnDisconnect();        // Feche socket/dispositivos antes de sair
        m_fExiting = true;     // Ativa indicador de saída
     }
     else
        EndDialog(m_hWnd, 0); // Sair se dispositivos fechados
}      // OnCancel
```

Na rotina OnDisconnect() o socket é fechado, os dispositivos são resetados e os flags m_fOutClosing e m_fInClosing são ativados para indicar que os dispositivos estão sendo fechados e que OnWomDone() e OnWimData() não devem mais preparar novos buffers para reutilização.

OnWomDone() decrementa m_iCountOut toda vez que um buffer é retornado por waveOut. Quando o contador chega a zero o dispositivo waveOut é fechado resultando na mensagem MM_WOM_CLOSE que é tratada por OnWomClose(). Em OnWomClose() e em OnWimClose() se ambos os dispositivos são marcados

como fechados, e existe pedido de término da aplicação, EndDialog() é chamado para finalizar a aplicação.

```
void OnDisconnect() {
    if (m_hWaveOut != 0) { // Se o dispositivo de reprodução está aberto
        m_fOutClosing = true;  // Ativa flag que disp está fechando

        WSAAsyncSelect(m_Socket, m_hWnd, 0, 0); // Desabilita notificações
        closesocket(m_Socket);  // Fecha socket
        // Interrompe dispositivo de saída e retorna buffers para a aplicação.
        waveOutReset(m_hWaveOut);
        // Necessário pois podemos estar em modo delay
        if (m_iCountOut == 0)
            waveOutClose(m_hWaveOut);  // Fecha dispositivo
    }
    if (m_hWaveIn != 0) { // se o dispositivo de entrada está aberto
        m_fInClosing = true;  // Ativa flag que dispositivo está fechando
        // Interrompe dispositivo de entrada e retorna buffers para a aplicação.
        waveInReset(m_hWaveIn);
        if (m_iCountIn == 0)
            waveInClose(m_hWaveIn);  // Fecha dispositivo
    }
    return;
}     // OnDisconnect
```

Introdução de Melhoramentos

Vários melhoramentos poderiam ser introduzidos:

1) Enviar mais de um packet de dados redundantes de áudio nos pacotes.
2) Reduzir a qualidade dos dados dos pacotes redundantes, reduzindo a demanda por largura de banda.
3) Reduzir os tamanhos dos pacotes de dados quando transmitindo silêncio. Como um processo de conversação resulta numa transmissão *half duplex*, isto traria grande economia de banda.
4) Controle de desligamento do lado remoto.
5) Controle de volume de reprodução de áudio.
6) Controle de seleção do dispositivo de áudio (microfone, CD, etc.).

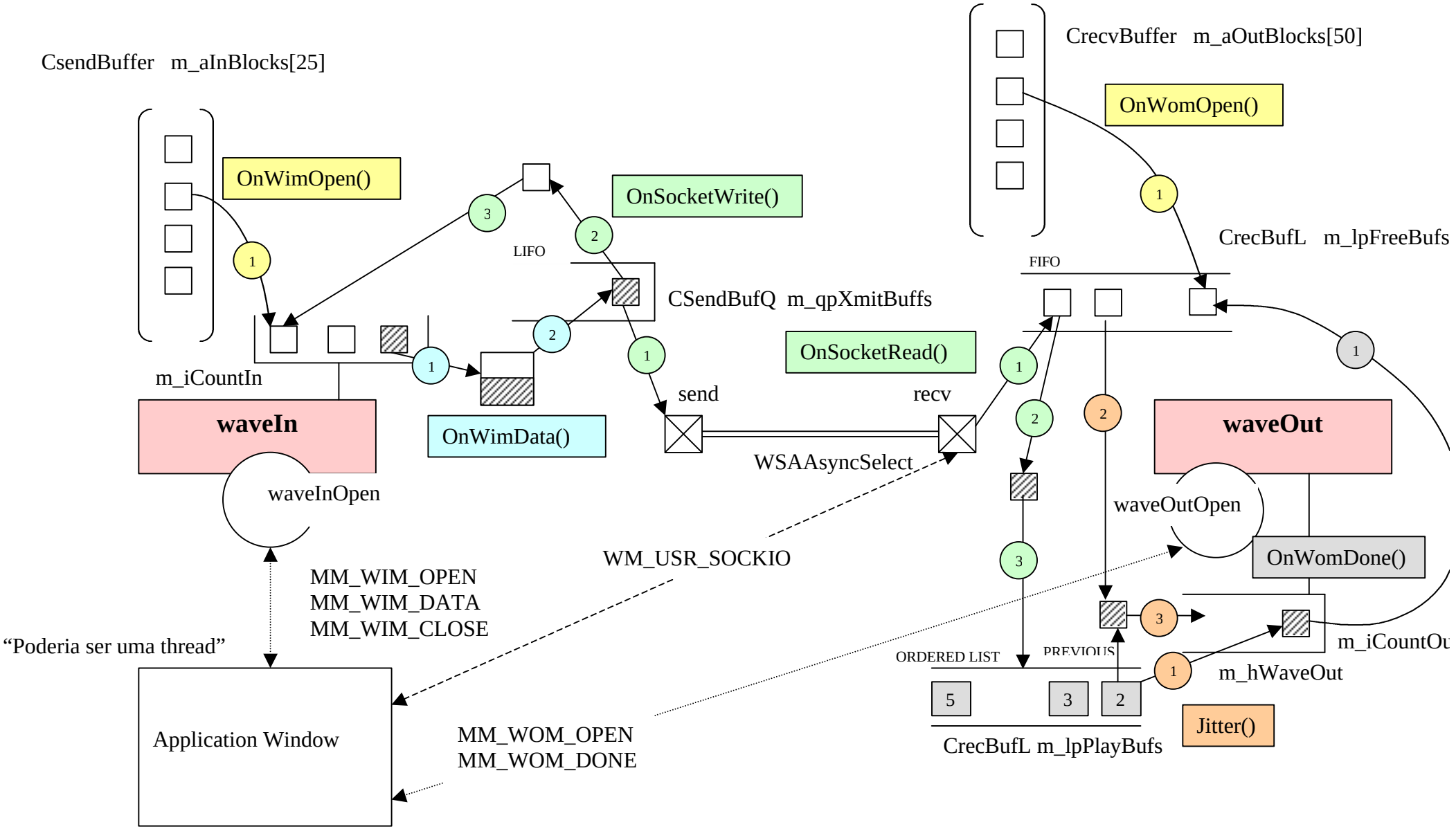


Figura 5: Diagrama Didático

Exercícios

1. Liste os métodos da classe std::queue<Tipo>

2. Liste os métodos da classe std::list<Tipo>

3. Faça um programa para demostrar as funcionalidades da classe queue.

4. Construa um programa para demonstrar as funcionalidades da classe list.

5. Marque nos programas os pontos onde:

Blocos duplicado são rejeitados	
Gaps de blocos são ignorados se NumBlocos disponíveis < 5	
Blocos são ajuntados antes de serem tocados se NumBlocos < 10	
O bloco de ordem N é recuperado a partir do bloco de ordem N+1 quando o bloco de ordem N é perdido	
Blocos chegando em qualquer ordem são reordenados na fila de chegada	

6. Ao invés de usar um I/O assíncrono para receber e enviar packets via sockets, use um mecanismo síncrono, isolando as funções de transmissão e recepção em threads dedicadas.

7. Para que você usaria este programa em aplicações práticas ?

Bibliografia

[Dandass 2001] Yogi Dandass, Streaming Real-Time Audio, Windows developer's journal, January 2001

[Microsoft Msdn] Microosft – Msdn - Waveform audio overview