Tutorial Básico do OpenH323

Versão 1.0-2
Por Vladimir Toncar ("*Para ajudar você a entender melhor o OpenH323.*")
Traduzido por Cesar Marcondes - 04/04/03 / Atualizado - 28/04/03

Índice

1 - Introdução

- Bem-vindo ao Tutorial Básico do OpenH323. O objetivo deste trabalho é descrever e explicar diversas técnicas de programação relacionadas com as bibliotecas PWLib e OpenH323, começando pelas mais básicas e mais tarde discutindo um dos tópicos mais avançados.
- A aplicação que acompanha este tutorial foi projetada com o compromisso de manter a simplicidade e funcionalidade. Ela tem que ser fácil de estudar. Por causa disto, ela deve focar na apresentação das técnicas básicas e deixar de lado as características adicionais.
- O resultado é uma aplicação simples chamada oh323tut. Ela trabalha como um player de áudio WAV baseado no OpenH323: ou seja, voce chama esta aplicação com um telefone H.323 e ela tocará um arquivo WAV através da conexão H.323.
- O tutorial mostra como usar as bibliotecas PWLib e OpenH323 para:
  - derivar uma classe "application";
  - segmentar as opções de linha de comando;
  - derivar uma classe "H.323 endpoint";
  - inicializar o terminal (endpoint);
  - otimizar o comportamento do terminal pelo uso de métodos virtuais;
  - ler dados de áudio do arquivo WAV e manipular a temporização corretamente.
- Do leitor é recomendado que tenha conhecimento básico de H.323 (terminais, aliases, sinalização de chamada básica). Ele também deveria estar familiarizado com C++, sendo um pouco mais específico, é preciso ter um bom entendimento do conceito do polimorfismo.
- A aplicação do tutorial foi desenvolvida para Linux. Embora esta também deva funcionar em vários sistemas baseados em Unix e algumas poucas mudanças são necessárias para se portar a aplicação para o ambiente Win32.
- O tutorial inicia com uma breve descrição de como compilar as bibliotecas Pwlib, OpenH323 e a aplicação tutorial (oh323tut) em um sistema Linux/Unix na [Seção 2].
- [Seção 3] contém um breve manual do usuário para a aplicação oh323tut. Se você deseja somente testar a OpenH323 e precisa do oh323tut como um terminal recebendo chamadas feitas a partir do ohphone, então as Seções 2 e 3 são tudo o que você precisa.
- [Seção 4] lida com a classe "application" do oh323tut's. Esta seção explica como derivar a classe "application" da classe "PProcess" da biblioteca PWLib, além de detalhar como segmentar aquelas opções presentes na linha de comando, etc.
- [Seção 5] descreve a classe "H.323 endpoint" e cobre tópicos como a derivação desta classe, a inicialização da classe, a inicialização do terminal (endpoint) - no caso, o terminal fica "ouvindo" em uma determinada porta obtida, configuração dos aliases, a escolha de codificadores, etc. Também é explicado como usar os métodos virtuais e alterar o comportamento da classe "endpoint".
- [Seção 6] explica como ler os dados de áudio do arquivo WAV usando a classe "channel". Esta é a seção mais complexa deste tutorial , pois ela fala do problema da temporização adaptativa.

1.1 - Convenções Tipográficas

- As seguintes convenções tipográficas serão usadas neste texto:
  - nomes de arquivos, trechos de código, entrada de linha de comando serão escritos na fonte Courier. ex.: `pconf.h`, `make bothdepend`;
  - nomes de variáveis, nomes de funções, nomes de métodos, e nomes de aplicações serão escritos em itálico. ex.: *fileName, main(), ohphone*;
  - nomes de classes são escritas em fonte normal. ex.: PProcess, MyEndPoint.

1.2 - Direitos Autorais

- O Tutorial Básico OpenH323 tem seus Direitos Autorais Reservados (c) 2002-2003 para Vladimir Toncar, Ph.D. Nenhuma parte deste tutorial pode ser reproduzida sem a autorização prévia do autor. A aplicação que acompanha este tutorial (oh323tut) está publicado segundo os termos da Licença Pública Mozilla Versão 1.1. você pode obter uma cópia da Licença em http://www.mozilla.org/MPL/.

- Todas as logomarcas registradas são de propriedade de seus respectivos donos.

1.3 - Informações de Contato

- Por favor, envie comentários a respeito deste tutorial para oh323tut@toncar.cz.
  No caso de comentários referentes a tradução deste tutorial, envia email para cesar@voip.nce.ufrj.br.

2 - Compilando a Aplicação

- Primeiramente, é preciso obter as bibliotecas PWLib e OpenH323. Não explicaremos em detalhes como obter e como compilar estas duas bibliotecas, dê uma olhada no site do OpenH323 para instruções mais detalhadas.
- Nesta seção é apresentada uma receita rápida para construir binários sem o uso das bibliotecas "compartilhadas" em uma plataforma Linux/Unix. Nós preferimos criar código executável "não-compartilhado" aqui porque eles são ligeiramente mais fáceis de usar. Por exemplo, você pode transferir os executáveis para outra máquina sem precisar instalar as bibliotecas PWLib e OpenH323.
  - 1. Coloque as bibliotecas PWLib e OpenH323 em seu diretório home, ex. PWLib fica no `~/pwlib` e o OpenH323 fica no `~/openh323`.
  - 2. Vá para o diretório `~/pwlib` e execute
    - ```
      make bothdepend
      make bothnoshared
      ```
      ```

        - 3. Vá para o diretório `~/openh323` e execute
            - `make bothdepend`
              `make bothnoshared`

- Se o seu compilador for o gcc 2.96, você precisará preferencialmente de 256 MB de RAM (mais 128 MB de swap) ou no pior caso, de pelo menos 196 MB de RAM (muito mais swap) para construir as versões de debug e otimizada do OpenH323 em um tempo razoável. Se você não tiver a memória RAM suficiente e ainda deseja trabalhar com o software, tente compilar somente a versão debug, usando a opção `make debugnoshared`.
- Se o seu gcc é de uma versão diferente do que a 2.96, ex. 2.95 ou 3.2, os requisitos de memória são menores. Por exemplo, com gcc 3.2 voce deve estar habilitado a contruir tanto as versões debug e otimizada de seu OpenH323 com 128 MB de RAM e 128 MB de espaço de swap.

- Em seguida, faça o download do arquivo oh323tut-1.0-1.tgz (ou quaisquer versões mais novas, veja a página de downloads) para um diretório e descompacte o arquivo usando o comando:
    - `tar -zxvf oh323tut-1.0-1.tgz`

- Entre no diretório oh323tut-1.0-1 e crie a aplicação usando os comandos `make bothdepend; make bothnoshared`. Com isto você obterá tanto a versão release (otimizada) quando a debug. Se você não for precisar de ambas, use o comando `make optnoshared` ou `make debugnoshared`.
- Se você tiver as bibliotecas PWLib e OpenH323 localizadas em algum lugar diferente de `~/pwlib` and `~/openh323`, então você deve configurar duas variaveis de ambiente (`PWLIBDIR` and `OPENH323DIR`) antes de compilar. Os comandos para estas variáveis de ambiente são:
    - `export PWLIBDIR=/outro/lugar/pwlib`
      `export OPENH323DIR=/outro/lugar/openh323`
- Os arquivos objeto (`*.o`) e os executáveis oh323tut (a versão release e debug, respectivamente) serão criados nos subdiretórios `obj_linux_x86_r` e `obj_linux_x86_d`. Os nomes destes subdiretórios podem ter diferença dependendo da plataforma que for compilado o sistema, como no caso de não estar sendo usado Linux ou o processador não for compatível com o x86.
- Os executáveis produzidos pela compilação com opção "*bibliotecas não compartilhadas - noshared*" já contém todo o código necessário da PwLib e OpenH323. Eles entretando dependem de outras bibliotecas dinâmicas também - como a biblioteca standard C, standard C++, biblioteca pthreads, etc. Se você quiser verificar a lista completa de dependências, execute o comando `ldd oh323tut`.

3 - Usando a Aplicação

- Esta seção é um breve "manual de usuário" da aplicação *oh323tut*.

3.1 - Opções da linha de comando

- -f *foo* (--file *foo*)
  - Lê os dados de áudio de um arquivo WAV "*foo*". O arquivo deve ter o seguinte formato: PCM, mono, taxa de amostragem de 8000 Hz, tamanho da amostra de 16 bits. Este é um parâmetro obrigatório na linha de comando.

- -g *addr* (--gatekeeper *addr*)
  - Configura o endereço do gatekeeper em *addr*.

- -G *id* (--gatekeeper-id *id*)
  - Configura o identificador do Gatekeeper em *id*.

- -h (--help)
  - Imprime esta mensagem de ajuda e sai do programa.

- -n (--no-gatekeeper)
  - Não registrar no Gatekeeper.

- -o *file* (--output *file*)
  - Escreve um arquivo de saída em *file* com o trace das mensagens de debug.

- -p *portnum* (--port *portnum*)
  - Ouvir as "chamadas" na porta TCP especificada em *portnum*. O valor default é 1820, para permitir a coexistência simples com o *ohphone* ou outro software de telefonia H.323 no mesmo computador.

- -t (--trace)

        - Habilita as mensagens de trace. Pode ser usado múltiplas vezes para aumentar o detalhamento da saída do trace.

    - -u *userid* (--user *userid*)
        - Configura o alias do usuário em *userid*. O valor default é o nome de login do usuário corrente. A partir da versão 1.0-2 do oh323tut, esta opção pode ser usada mais do que uma vez para configurar vários aliases.

    - Se nem -g muito menos -G forem colocados como parâmetro, o padrão é tentar descobrir um gatekeeper com qualquer gatekeeper ID. Use a opção -n se quiser desativar este comportamento.

3.2 - Exemplos

3.2.1 Um Único Computador, Nenhum Gatekeeper

- Para testar o `oh323tut` com `ohphone` ou `simph323` em um único computador, sem o uso de um gatekeeper, você precisa invocar o `oh323tut` com o seguinte comando:
    - `./oh323tut -f audio.wav -n -u 320`
- Quando configurado com estes parâmetros, `oh323tut` irá "ouvir" as conexões que chegarem na porta TCP 1820. O arquivo de áudio `audio.wav` deve estar no formato requisitado. O alias "apelido" do usuários será configurado como 320.
- Para chamar o oh323tut a partir do simph323, use o seguinte comando (em outra janela de shell):
    - `./simph323 -n -u 321 320@127.0.0.1:1820`

3.2.2 Registrando o oh323tut com um Gatekeeper

- Suponha que você esteja executando um gatekeeper no endereço IP 10.1.2.3. Inicie a *oh323tut* assim:
    - `./oh323tut -f audio.wav -u 320 -g 10.1.2.3`

- Para usar o ohphone como um terminal de chamada, inicie ele conforme abaixo:
    - `./ohphone -l -g 10.1.2.3 -u 321`

- Uma vez que você tiver o ohphone executando, você pode iniciar uma chamada usando o comando c 320 ("call 320").

4 - A Classe OH323Tut

4.1 Declaração

- Cada programa que é baseado na PWLib deve ter uma instância de uma classe descendente de PProcess ou alternativamente de PServiceProcess (que por si só é descendente de PProcess). Em nossa aplicação tutorial, esta classe é chamada OH323Tut. A classe é declarada no arquivo oh323tut.h (linhas 33–42). O método virtual Main() (linha 40) é o local onde nós colocaremos o código que normalmente (em um programa C/C++ que não use PWLib) estaria dentro da função main() do programa.

- A declaração do OH323Tut é relativamente simples. Além do método *Main()*, existem apenas um constructor, o destrutor, o método *printHelp()* (declarado na linha 41) que imprime informações sobre o uso do programa, e a macro PCLASSINFO. A tarefa da macro (linha 35) é inserir métodos de tipagem em tempo-real que são obrigatórios para cada membro da hierarquia de classe PWLib. Se você estiver interessado, a macro é definida no arquivo `pwlib/include/ptlib/object.h`.

4.2 Implementação

- O construtor da classe OH323Tut está no arquivo `oh323tut.cxx` na linha 58. A única tarefa do construtor é passar parâmetros para o construtor da classe pai (PProcess). Os parâmetros são, nesta ordem, o autor da aplicação, o nome da aplicação, os números de versão maior e menor da aplicação, o status do código fonte (possíveis valores são AlphaCode, BetaCode, e ReleaseCode), e o número da aplicação (build). Nós configuramos os quatro últimos parâmetros usando constantes que estão definidas no arquivo `pconf.h` (linhas 50–53).
- Um pesquisa nos fontes do *oh323tut* deve mostrar que nenhum dos arquivos contém explicitamente uma função *main()*. A *main()* é gerada pela macro PCREATE_PROCESS (arquivo `oh323tut.cxx`, linha 56). Nas plataformas Unix, a macro é definida assim:

```
#define PCREATE_PROCESS(cls) \
    int main(int argc, char ** argv, char ** envp) \
      { PProcess::PreInitialise(argc, argv, envp); \
        static cls instance; \
        return instance._main(); \
      }
```

- O código inserido pela macro é um pouco diferente em cada plataforma suportada pela PWLib, mas a variante Unix acima deve dar uma idéia sobre o que está acontecendo neste contexto. O *main()* simplesmente realiza alguma inicialização, aloca uma instância da classe *cls* (no nosso caso OH323Tut) e chama o método *_main()* daquela instância. O

_main() por sua vez chama o método virtual *Main()*. A macro ajuda a você criar programas multi-plataforma de uma maneira simples — com uma única linha que é mais elegante do que diversas #ifdef.

- Continuando a descrição do método *Main()* do OH323Tut (arquivo `oh323tut.cxx`, linhas 92–168). Nós pularemos as linhas 94–97 (retornaremos nelas mais tarde) e discutiremos a segmentação (parsing) de argumentos da linha de comando.
- Na linha 98, nós criamos uma instância da classe PConfigArgs:

```
PConfigArgs args(GetArguments());
```

- Este é o "dialeto" de programação usado em muitos programas PWLib. *GetArguments()* é um método de PProcess que retorna a lista de argumentos dado para o programa a partir da linha de comandos.
- O comando subsequente na linha 100 diz ao objeto *args* quais opções de linha de comando nós precisaremos segmentar. Cada opção é dada em sua forma curta e longa, ambas sem o sinal '-' na frente. Então, por exemplo, a substring "h-help." significa que a requisição de help irá aparecer tanto com o parâmetro `-h` ou `--help` na linha de comando. As substrings que descrevem as opções com uma string opcional (ex. `-f` ou `--file` tem que ser seguido por um nome de arquivo) terminam com ":", ex. "f-file:". A substring que descreve opções sem uma string opcional terminam com um ponto, ex. "h-help.". Observe que os argumentos 't' e 'o' serão somente incluídos se o programa compilar com o tracing habilitado (que é padrão tanto para o binário *opt* quanto para o *debug*).
- A única opção de comando de linha obrigatória para a aplicação *oh323tut* é `-f` (ou `--file`). Por causa disto, as linhas 116–120 mostram a ajuda e terminam o programa se a opção 'f' estiver ausente ou se 'h' ('-help') estiver presente.
- As linhas 122–126 configuram o nível de trace e opcionalmente o arquivo de saída de trace. As linhas 128-156 processam individualmente as opções de linha de comando. Nós usaremos uma instância da classe ProgConf (declarada no arquivo pconf.h, linhas 32–47) para armazenar as informações obtidas da linha de comando.
- As três opções 'n', 'g', e 'G' estão relacionadas com o registro do terminal com um gatekeeper. Nós primeiramente configuramos o *progConf.gkMode* para `progConf::RegisterWithGatekeeper` — sendo este o comportamento padrão no caso de nenhuma das três opções de gatekeepers sejam usadas (linha 130). A ordem na qual as opções são processadas importa (linhas 132–145). Se o usuário colocar tanto a opção `-n` quanto a opção `-g/-G`, então a `-n` é declarada inválida. Tanto 'g' quanto 'G' configuram *progConf.gkMode* para o valor `ProgConf::RegisterWithGatekeeper` e ambas requerem uma string opcional que será armazenada em `progConf.gkAddr` ou em `progConf.gkId`, respectivamente.

- A linha 147 armazena o nome do arquivo em *progConf.fileName*. Observe que não é necessário testar para a presença da opção 'f' — nós já realizamos esta na linha 116. A linha 149 testa a presença da opção 'p'. Se a opção estiver presente, o número da porta será associado no *progConf.port*. A linha 152 realiza um teste de validação simples no número da porta — se o número for zero, nós reconfiguramos novamente para o valor padrão.
- A linha 155 testa a presença da opção 'u' (user alias). O usuário pode colocar diversos aliases na linha de comando e a PString será retornada pela expressão `args.GetOptionString('u')` contendo cada alias em uma linha separada. O método *Lines()* transformará a string em um vetor contendo os aliases individuais (linhas) e então atribuiremos este vetor ao *progConf.userAliases*.
- A linha 159 cria uma instância da classe MyEndPoint. Se a inicialização do endpoint (linha 161) tiver sucesso, a execução da thread principal da aplicação será bloqueada na linha 164. O objeto *terminationSync* é uma instância da classe PSyncPoint (veja linha 36). PSyncPoint é de fato um semáforo que fica inicialmente bloqueado, de forma que a thread que chamar o método *Wait()* ficará bloqueada até que outra thread chame o método *Signal()*.
- Queremos parar a aplicação quando o usuário pressionar Ctrl-C. Para fazer isto, nós usamos o objeto *terminationSync* juntamente com a manipulação de sinais em Unix. O manipulador de sinais está definido nas linhas 38–51 e nós registramos a ele os sinais SIGINIT e SIGTERM nas linhas 94–97. Quando o manipulador de sinais *signalHandler()* receber qualquer um dos dois sinais, ele simplesmente chama o *terminationSync.Signal()*. Este desbloqueia a thread de execução que estava esperando na linha 164. E quando o método *Main()* terminar, todas as variáveis automaticamente criadas pelo método serão desalocadas. Isto inclui o objeto *endpoint* — seu destrutor fecha todas as conexões ativas (se existir alguma), e faz todo o possível para que o endpoint seja desligado. Descreveremos a classe endpoint na próxima seção.

5 - Terminal H.323 (H.323 Endpoint)

- A biblioteca OpenH323 implementa o comportamento de um terminal H.323 em uma classe chamada H323EndPoint. Este classe tem um grupo de métodos virtuais que permitem ao programador otimizar o comportamento do terminal . Os métodos virtuais também servem como uma interface de comunicação através do qual
a OpenH323 notifica a aplicação sobre importantes eventos - como a chegada de uma nova chamada, o fim de uma chamada, etc.

- A tarefa do programador é derivar uma nova classe sobre a H323EndPoint e reescrever um conjunto de métodos desta classe H323EndPoint. Uma observação para aqueles que são novos com o conceito de polimorfismo: Se você deseja reescrever o comportamento do método de uma classe descendente ela deve ter exatamente a mesma assinatura que na classe pai. Uma falha na realização disto significa que você está sobrecarregando o método, o que é diferente do que reescrevendo.

- Um item importante a se lembrar sobre o uso de métodos virtuais no OpenH323 é que o código interno dos métodos não deve demorar muito tempo para executar. Você precisa estar atento aos timeouts na negociação da conexão do H.323.
  A outra parte chamada deve desconectar a chamada se o seu endpoint falhar em responder em tempo por causa de um atraso em um método virtual. O método virtual deveria somente passar informação sobre o evento para o código da sua aplicação e retornar. Um bom exemplo disto é que nós jamais devemos esperar pela resposta do usuário dentro método *OnAnswerCall().*

5.1 Declaração

- Nossa aplicação tutorial declara a classe descendente de H323Endpoint no arquivo `ep.h`. A nova classe é chamada MyEndPoint (arquivo ep.h, linha 31). MyEndPoint precisa ter acesso a configuração da aplicação — para conhecer o número da porta a ser "ouvida", ou o alias do usuário, por exemplo. Por causa disto, MyEndPoint tem uma referencia para a classe ProgConf (arquivo `ep.h`, linha 34).
- A tarefa do método *Init()* do MyEndPoint (linha 38) é inicializar o terminal. Esta inicialização significa, entre outras coisas, que o terminal inicia a "escuta" de chamadas que estiverem chegando e (opcionalmente) se registra com o gatekepeer.
- As linhas 39–52 listam os métodos que o MyEndPoint precisa reescrever. Os nomes dos métodos são mais ou menos auto-explicativos. *OnConnectionEstablished()* é chamado toda vez que uma conexão tiver sido estabelecida com sucesso. *OnConnectionCleared()* é chamado quando uma conexão tiver sido fechada.
  OpenAudioChannel() é chamada quando o stack OpenH323 precisa (criar e) abrir um novo canal de áudio. O método *OnAnswerCall()* é chamado quando a stack precisa decidir se ela aceita uma chamada (isto é uma explicação muito simplificada, nós voltaremos a detalhar este método abaixo). Por último, *OnStartLogicalChannel()* é chamado toda vez que um canal lógico for aberto. Observe que existem outros métodos virtuais definidos no H323EndPoint mas nossa aplicação não precisará reescrever todos eles.

5.2 Implementação

5.2.1 Init()

- Vamos agora descrever os métodos no arquivo `ep.cxx`. Nós inicialmente focaremos no método *Init()* de MyH323Endpoint (linhas 76–138). A tarefa deste método é inicializar o endpoint H.323, ex. configurar o alias do usuário, configurar os codificadores, iniciar o "listening" para as chamadas que estão chegando, e opcionalmente registrar com um gatekeeper. Nós descreveremos estes passos individualmente em mais detalhes abaixo.

- O código nas linhas 79 e 84 primeiramente checa se o vetor de strings *progConfig.userAliases* está não-vazio e nesse caso, ele insere os aliases do terminal. Se não tiver sido configurado nenhum aliases, o alias padrão é nome de login do usuário.
- Quando usarmos *SetLocalUserName*(`str`), você deve estar atento que ele primeiro limpa a lista de aliases do endpoint e então configura `str` como o primeiro (e único) alias. Se você precisar dar ao endpoint diversos aliases, use a chamada *SetLocalUserName()* para o primeiro alias e *AddAliasName()* para o segundo alias, o terceiro, etc.
- A classe endpoint armazena os aliases como uma lista de strings. Quando ela construir as mensagens H.225, as strings de alias serão incluídas com o tipo AliasAddress — tanto como *dialedDigits* (se a string contiver somente os caracteres "0123456789*#") ou como *h323-ID*.
- As linhas comentadas 86–88 mostram como desabilitar diversos recursos H.323, com o procedimento Fast Connect, o Tunelamento H.245 e o "H.245 dentro do Setup".
- As linhas 91–99 especificam quais codificadores o MyEndPoint deverá usar. A ordem das chamadas de *SetCapability()* configuram a prioridade dos codificadores. Nossa aplicação terá o codificador Speex com uma taxa de 8000 bps (chamado de SpeexNarrow3) como o codificador preferencial, o GSM 06.10 será o segundo preferencial, e o G.711 uLaw será o terceiro, etc. Nós também configuraremos o número de frames que o codificador usará na montagem dos pacotes RTP tanto para o Speex quanto para o GSM. Casualmente, os dois codificadores usam frames do mesmo tamanho, 20 milisegundos. Desta forma, se um canal lógico usar o GSM 06.10, um pacote RTP transmitido carregará cerca de 80 milisegundos de áudio (quatro frames). No caso do Speex, o pacote conterá 100 milisegundos de áudio. O site do OpenH323 tem uma tabela com os dados sobre o tamanho do frame, duração do frame e consumo de banda passante para um conjunto de codificadores.site has a table with data about frame size, frame duration, and bandwidth for a number of codecs.
- A linha 101 insere na tabela de capacidades todas as capacidades disponíveis de DTMFs. Se a aplicação executar com o nível de trace 1 ou maior, a linha 103 mostrará a tabela de capacidades do endpoint na saída padrão do trace.
- As linhas 106–115 alocam e inicializam um objeto *"listener"* da classe H323ListenerTCP. A função deste objeto é "ouvir" as conexões H.323. O construtor tem três parametros — o primeiro é uma referência para o H323EndPoint (ou seu descendente), o segundo parâmetro é o endereço da interface de rede o qual nós gostaríamos de "ouvir". O terceiro parâmetro é o número da porta (nós usaremos o valor armazenado no *progConf.port*). Se o valor INADDR_ANY estiver na variável *addr* significa gostaríamos de ouvir todas as interfaces disponíveis. Observe que o programador é o responsável pela desalocação do objeto *listener* se ele falhar ao tentar iniciar apropriadamente (ex. *StartListener()* retorna FALSE).
- As linhas 118–135 tratam do registro do endpoint com um gatekeeper. Se nós não quisermos registrar, simplesmente configuramos a flag de registro com sucesso e vamos em frente (o `case` mostrado na linha

121). Se nós precisarmos registrar, nós chamaremos o método *UseGatekeeper()* do H323EndPoint. Este método chamará um destes quatro métodos, cujos nomes são *DiscoverGatekeeper(), LocateGatekeeper(), SetGatekeeper(), e SetGatekeeperZone()*, usando o critério mostrado na tabela abaixo:

Endereço do Gatekeeper Disponível	Identificador do Gatekeeper Disponível (GK Zone)	Método Chamado
Não	Não	*DiscoverGatekeeper()*
Não	Sim	*LocateGatekeeper()*
Sim	Não	*SetGatekeeper()*
Sim	Sim	*SetGatekeeperZone()*

-
- O primero pacote que for enviado pelo endpoint durante o processo de registro carrega a mensagem H.225 RAS Gatekeeper Request (GRQ). Se o endereço do gatekeeper não for conhecido (ex. através de *DiscoverGatekeeper()* e *LocateGatekeeper()*, respectivamente), a mensagem GRQ será enviada usando multicast. O endereço multicast associado aos gatekeepers para registro é 224.0.1.41. Se o nome da zona estiver disponível (*LocateGatekeeper()* e *SetGatekeeperZone()*, respectivamente), o endepoint coloca opcionalmente o identificador no campo *gatekeeperIdentifier* do GRQ. Se o registro falhar, a linha 133 gera uma mensagem simples de erro e a *Init()* retorna com valor falso, do contrário o endpoint terá sido inicializado com sucesso, e o *Init()* retornará verdadeiro.

5.2.2 Métodos Virtuais

- Nós agora focaremos na implementação dos métodos virtuais.

- **OnConnectionEstablished()**
  - O método virtual MyEndPoint::*OnConnectionEstablished()* (arquivo `ep.cxx`, linhas 45–49) é chamado quando a conexão H.323 tiver sido estabelecida com sucesso. Nossa aplicação tutorial apenas usa este método como gerador de saída para o trace, mas nós podemos imaginar que o *OnConnectionEstablished()* é bastante importante para aplicações do mundo real. Como por exemplo, se nós precisarmos realizar algum tipo de bilhetagem, esta callback denota o início da duração de uma chamada sendo bilhetada.
  - *OnConnectionEstablished()* tem dois parâmetros. O primeiro é uma referência para um objeto H323Connection (o objeto está bloqueado, você terá acesso exclusivo a ele). O segundo parâmetro (const PString & *token*) é uma referência a uma string que identifica unicamente uma conexão em particular dentro do endpoint OpenH323. Não cometa erros chamando os OpenH323 call tokens de H.323 call identifiers. Os call tokens são internos a biblioteca OpenH323 e eles não aparecerão em

quaisquer mensagens enviadas pelo endpoint. Os call tokens são construídos usando o nomes dos hosts e os números das portas e são portanto legíveis — isto pode ajudar quando você estiver debugando o seu programa. Toda vez que sua aplicação precisar associar o seus dados com as chamadas H.323 individualmente, os call tokens serão os links naturais.

- **OnConnectionCleared()**
  - O método virtual MyEndPoint::*OnConnectionCleared()* (arquivo `ep.cxx`, linhas 54–58) serão chamados quando uma conexão tiver sido fechada. Seus parâmetros são os mesmos daqueles de *OnConnectionEstablished()*. Se você precisar saber a razão pela qual a chamada foi finalizada, você pode obter isto ao chamar *connection.GetCallEndReason()*. O tipo de dado retornado por este método é enum CallEndReason — por favor verifique o arquivo `openh323/include/h323con.h` para todos os possíveis valores.
- **OnAnswerCall()**
  - O terceiro método virtual que nós lidaremos é o MyEndPoint::*OnAnswerCall()* (arquivo `ep.cxx`, linhas 63–71). Este callback é o local onde o programador da aplicação decidirá se o stack OpenH323 aceitará ou rejeitará uma chamada que chegou. O método tem quatro parâmetros. O primero parâmetro é uma referência a um objeto H323Connection, enquanto o segundo é uma PString dando o nome (descrição textual) do chamador. O terceiro parâmetro é uma constante de referência com a PDU de Setup Q.931/H.225 que o endpoint local recebeu do chamador. O quarto parâmetro é uma referência para a PDU PDU que deverá ser enviada se a chamada for aceita.
  - *OnAnswerCall()* não é a única callback que é usada quando estamos decidindo pela aceitação ou rejeição de uma chamada. A classe H323EndPoint tem outro método virtual, *OnIncomingCall()* (veja `openh323/include/h323ep.h` para verificar seu protótipo exato). Quando um ponto remoto quiser estabelecer uma conexão H.323, ele inicialmente envia a PDU de Setup. A stack OpenH323 local responderá com a PDU Call Proceeding e então invocará a callback *OnIncomingCall()*. Se a *OnIncomingCall()* retornar TRUE, o endpoint local enviará a PDU Alerting e então será chamado o método *OnAnswerCall()*. Então se você já souber no momento que o *OnIncomingCall()* for chamado que você não poderá aceitar a chamada (ex. a sua aplicação está carente de um certo recurso em particular), você pode rejeitar a chamada pelo retorno de FALSE. De outro modo, você não precisa reescrever *OnIncomingCall()* — e o seu comportamento padrão é retornar TRUE - e deixar a manipulação para o *OnAnswerCall()*.
  - Nossa aplicação tutorial sempre aceitas as chamadas que estiverem chegando, então a chamada MyEndPoint::*OnAnswerCall()* simplismente retornará `H323Connection::AnswerCallNow`. Se você desejar recusar a chamada, o valor retornado deveria ser `H323Connection::AnswerCallDenied`. Como mencionado no topo desta seção, se você não puder decidir imediatamente se aceita uma chamada (o que seria o caso em muitos programas do mundo real), não seria correto esperar dentro de *OnAnswerCall().* Ao invés disto, o *OnAnswerCall()* deveria apenas notificar a aplicação sobre a chegada de uma nova chamada e então retornar

`H323Connection::AnswerCallPending`. A stack OpenH323 então enviará a PDU Alerting e a aplicação poderá aceitar ou rejeitar a ligação mais tarde, usando o método *AnsweringCall()* da classe H323Connection.

- **OpenAudioChannel()**
    - MyEndPoint::*OpenAudioChannel* (arquivo `ep.cxx`, linhas 143–161) é outro método virtual que é importante para nossa aplicação tutorial. Esta callback permite que a aplicação configure a origem e o destino do áudio para alguma coisa a mais do que a placa de som. Três dos quatro parâmetros do *OpenAudioChannel()* são importantes para nossa aplicação: H323Connection & *connection* é a referência para a conexão da qual nós queremos criar o canal de áudio. BOOL *isEncoding* nos diz a direção do dados de áudio. Se *isEncoding* for verdadeiro, o canal de áudio agirá como uma fonte de dados de áudio para ser codificado e transmitido para o ponto remoto, se for o contrário o canal de áudio será usado como um destino para o áudio que está chegando. O terceiro importante parâmetro é o H323AudioCodec &*codec* que nos dá acesso ao objeto do codificador que tanto irá codificar o áudio de saída ou decodificar o áudio de entrada.
    - Vamos agora descrever o código interno do *OpenAudioChannel().* A linha comentada 148 é um exemplo de como desabilitar a detecção de silêncio. A linha 149 testa o valor de *isEncoding*. Se ele estiver em verdadeiro (ex. nós estamos tratando com o áudio de saída), nós vamos criar um objeto da classe WavChannel. O construtor da classe WavChannel precisa de dois parâmetros — o primeiro é o nome de uma arquivo WAV enquanto que o segundo parâmetro é uma referência para o objeto H323Connection que irá usar o objeto channel (veja a seção 6 para uma discussão detalhada de canais de áudio). Tendo então criado o objeto channel, nós anexamos a ele o codec (linha 152). O valor `true` passado como segundo parâmetro do *AttachChannel()* significa que o codec poderá desalocar o objeto channel quando ele não for mais necessário. O *AttachChannel()* retorna um valor booleano indicando se a operação teve sucesso e nós usaremos este booleano como o valor de retorno do *OpenAudioChannel()*. O código para o caso quando o *isEncoding* é falso (áudio de chegada) é quase idêntico. A única diferença é que nós criamos uma instância da classe NullChannel ao invés da WavChannel.
- **OnStartLogicalChannel()**
    - O método virtual do endpoint *OnStartLogicalChannel()* é chamado quando o stack OpenH323 tiver iniciado com sucesso uma thread responsável por tanto transmitir ou receber dados de áudio ou vídeo. Nossa aplicação tutorial somente usará áudio, então nós teremos duas threas de canais lógicos para cada chamada. Nós vamos usar o método virtual MyEndPoint::*OnStartLogicalChannel()* (arquivo `ep.cxx`, linhas 166–185) para gerar informação de trace sobre o codificador (capability) usado tanto para os dados de entrada quanto para os de saída. O método é chamado uma vez para cada channel, desta forma se a saída do trace para uma chamada em particula tiver uma e somente uma linha contendo "Started logical channel...", nós saberemos que será preciso procurar por problemas de negocição H.323, ex. codificadores com erro ou banda passante alocada insuficiente.

6 - Canais de Áudio

- Nesta seção, descreveremos as classes do canal de áudio usado em nossa aplicação tutorial. O comportamento padrão do stack OpenH323 é ler (e escrever) dados de áudio a partir (ou para) uma placa de som. Nossa aplicação precisa reescrever este comportamento padrão e portanto nós teremos que definir nossos próprios canais de áudio.

- A aplicação *oh323tut* usa duas classes `channel`, ambas derivadas da classe PIndirectChannel: WavChannel (arquivos wavchan.h e wavchan.cxx) e NullChannel (arquivos nullchan.h e nullchan.cxx). Enquanto estiver estudando o código fonte, nós provavelmente veremos que a funcionalidade destas duas classes podem ser combinadas em uma única classe. Entretanto, nós temos que separar as duas classe para tornar mais fácil o entendimento. Isto também ajuda na enfase que sera dado ao fato do áudio de entrada e áudio de saída serem independentente um do outro.
- Quando definimos duas novas classes de channel, precisamos reescrever quatro métodos virtuais, *Close()*, *IsOpen()*, *Read()*, and *Write()*. Você encontrará os protótipos destes métodos no arquivo `wavchan.h` (linhas 46–49) ou em `nullchan.h`, respectivamente (linhas 45–48). O papel de cada um dos quatro métodos virtuais é fácil de entender. Uma vez que uma instância de channel seja criada, se espera que ela possa ser aberta. O método *IsOpen()* é usado para verificar que tudo aconteceu corretamente durante a criação do canal (ex. inicialização do dispositivo, abertura do arquivo), o *Close()* é chamado quando o canal não é mais necessário. E os métodos *Read()* e *Write()* são usados para ler/escrever dados a partir da instância do canal (channel).

6.1 WavChannel

- A tarefa da classe WavChannel é ler os dados de áudio de um arquivo WAV. A
  declaração da WavChannel inicia na linha 36 no arquivo `wavchan.h`. A classe
  precisa de quatro objetos membros. O membro *myConnection* é uma referencia para uma classe H323Connection — ele é necessário para, por exemplo, fechar a conexão quando nós alcançarmos o final do arquivo WAV. O PWAVFile *wavFile* é um objeto que nos permite ler o áudio a partir de um arquivo WAV. Os dois outros membros restantes, *writeDelay* e *readDelay* são ambos do tipo PAdaptiveDelay. Explicaremos os papéis destes adaptive delays (atrasos adaptativos) em mais detalhes abaixo.
- Vamos descrever agora a implementação dos métodos do WavChannel.

6.1.1 Construtor e Destrutor

- O construtor do WavChannel (arquivo `wavchan.cxx`, linhas 30–52) tem dois parâmetros formais: uma referência para o nome de um arquivo WAV e uma referência para um objeto H323Connection. Estes dois parâmetros são usados nos inicializadores dos objetos membros conforme mostra a linha 31. O construtor do PWAVFile tenta abrir o arquivo, portanto a primeira coisa que nós temos que fazer dentro do construtor é checar se o arquivo foi aberto com sucesso (linhas 33–38). Se o arquivo não for aberto, nós vamos escreveremos uma mensagem de erro, fecharemos a conexão H.323 e retornaremos do construtor.
- Nosso próximo passo dentro do construtor (linhas 39–48) é checar se o arquivo
  WAV tem os parâmetros requeridos, ex. se o formato dele é PCM, mono (somente um canal de som), a taxa de amostragem é de 8000 Hz, e o tamanho da amostra é de 16 bits. Se o arquivo não se encaixar nestes requisitos, faremos exatamente o mesmo procedimento de erro anterior, ex. vamos escrever a mensagem de erro e fecharemos a conexão H.323.

- A última linha no construtor é somente uma declaração PTRACE que anuncia a
  criação com sucesso do objeto WavChannel.
- A única linha de código no destrutor WavChannel (linhas 57–60) é novamente uma declaração PTRACE que informa sobre a remoção do objeto channel.

6.1.2 Close() e IsOpen()

- A implementação dos métodos *Close()* (linhas 65–68) e *IsOpen()* (linhas 73–77) do WavChannel é bastante simples — eles retornam valores verdadeiros obtidos dos métodos *Close()* and *IsOpen()* do *wavFile*, respectivamente.

6.1.3 Ações Necessárias nos métodos Read() e Write()

- Além do processamento normal de dados de áudio, tanto os métodos *Read()* e *Write()* devem tomar cuidado com relação a duas coisas:

  1. notificar o chamador sobre os resultados da operação de leitura/escrita;
  2. tomar conta da temporização corretamente.

- **Notificação sobre o resultado de operações**
- Para notificar o canal do usuário sobre os resultados das operações de leitura e escrita, é precisa fazer duas coisas. Primeiramente, nós precisamos configurar a variável membro *lastWriteCount* (no método *Write()*) ou *lastReadCount* (no método *Read()*) para o número de bytes escritos ou lidos com sucesso. Após retornar dos métodos *Read()* ou

Write(), este número pode ser obtido dos métodos do channel *GetLastWriteCount()* e *GetLastReadCount()*.

- A segunda dica é que os métodos *Read()* e *Write()* devem retornar true ou false dependendo dos requisitos preescritos em `pwlib/include/ptlib/channel.h`. O método *Write()* deve retornar true se ele conseguiu escrever todos os bytes que passarem por ele, ou do contrário false. O método *Read()* deve retornar true se pelo menos um byte foi lido, do contrário retorna false.

- **Temporização**

- Adicionalmente ao esquema de notificação de sucesso ou falha da operação de leitura/escrita, nós temos que tomar cuidado com a temporização. Quando usamos uma placa de som como fonte ou destino dos dados de áudio, a placa provê a temporização precisa para nós. Por exemplo, se nós lemos 80 amostras com a frequencia de amostragem de 8000 amostras por segundo, a operação de leitura vai terminar (quase exatamente) em 10 milisegundos a partir do final da leitura anterior. A temporização é essencial especialmente para o método *Read()*, pois ela influencia a qualidade de áudio recebido da outra parte comunicante (receptor). Até mesmo em endpoints que tenham buffer de compensação de jitter, nós devemos enviar os pacotes RTP o mais precisamente possível.
- No nosso caso, nós não estamos usando uma placa de som com as classes WavChannel ou NullChannel, embora o tempo gasto dentro do *Read()* ou *Write()* deve novamente corresponder a quantidade de dados lidos ou escritos. Para obter isso, nós precisamos adicionar algum tempo de sleep. Por exemplo, se a operação Write() é chamada e precisa escrever 480 bytes (ou 240 amostras — o que corresponde a 30 milisegundos) e o processamento precisa de somente, digamos, 1 milisegundo dentro do *Write()* e 1 milisegundo entre duas chamadas de *Write()* consecutivas, o tempo adicional "sleep" deve preencher os 28 milisegundos restantes para garantir que o tempo entre duas invocações consecutivas de Write() seja de 30 milliseconds.
- O problema com a função sleep em muitos sistemas (tanto no Unix/Linux e Windows) é que esta não é exata. Ele normalmente é arredondado em múltiplos de 10 milisegundos. Desta forma o arredondamente pode ficar entre 0 e 9 milisegundos e isto é ruim se considerarmos que uma ligação normal tenha tanto *Read()* quanto *Write()* que trabalham com 80, 160, ou 240 amostras, correspondendo a 10, 20, or 30 milisegundos, respectivamente. Se nós fizermos o comportamento como se o tempo de *sleep* fosse exato, acumularemos um grande quantidade de erros durante algumas chamadas consecutivas de *Read()* ou *Write()*. Portanto, precisamos usar um algoritmo de sleep adaptativo, de modo que mesmo se for causado erro de temporização durante uma chamada ao *Read()* ou *Write()*, este erro será compensado na chamada subsequente (ou chamadas). Deste jeito, o tempo de partida não será exato para cada pacote RTP individualmente, mas o intervalo médio entre os dois pacotes será bem mais próximo do valor exato.

- A PWLib implementa o algoritmo de sleep adaptativo em uma classe chamada PAdaptiveDelay. A classe usa um conceito de "tempo alvo". Quando o método *Delay(int time)* da classe PAdaptiveDelay é chamado pelo primeiro *time* (*time* está em milliseconds), o tempo alvo será configurado como o tempo atual mais um *time* em milisegundos. Durantes chamadas subsequentes de Delay(), o tempo alvo será simplesmente incrementado por *time* milisegundos. Após ajustar o tempo alvo, o algoritmo computa a diferença entre o tempo alvo e o tempo atual e pausa "sleep" esta diferença.
- O fato de usar o tempo alvo (um valor absoluto) nos ajuda a evitar o acumulo de erros de temporização. Suponha que ***Ti*** seja o tempo alvo na *i-esima* iteração, ***N*** seja o tempo atual (Agora) e ***e*** seja o erro de "sleeping". Se, na *i-esima* iteração, o sleep tomar ***(Ti - N + e)*** de tempo, a i-esimo iteração termina no tempo ***Ti + e***, ao invés de (o ideal) ***Ti***. Na próxima iteração (i+1)-esima, o tempo corrente (***N***) será aproximadamente igual a ***Ti + e***, de forma que a duração do sleep deve ser computada como ***Ti+1 - N = Ti+1 - Ti - e***, assim o erro sera compensado. Novamente, o sleep na (i+1)-esima iteração não será exato e assim ele será corrigido na (i+2)-esima iteração, e assim por diante. Dessa forma, a duração média de uma iteração ficará muito próxima do tempo ideal.

6.1.4 Write()

- A classe WavChannel tem como intenção primária ler dados de um arquivo de áudio. Por causa da natureza de nossa aplicação, o método *Read()* é mais importante do que o *Write()*. De fato, WavChannel::*Write()* nunca será chamado diretamente de nossa aplicação, porque o método MyEndPoint::*OpenAudioChannel()* associa uma instância de WavChannel a uma thread responsável pelo áudio de saída. Nós entretanto temos que implementar WavChannel::*Write()* (arquivo `wavchan.cxx`, linha 82–88). Agora demonstraremos os poucos passos que são necessários para cada método *Read()* ou *Write()* de channel.
- Nosso método WavChannel::*Write()* simplesmente ignora quaisquer dados que passarem por ele, mas ele tem que fingir tratar os dados do buffer com se estes tivessem sido escritos com sucesso. Este comportamento é de fato o mesmo que direcionar para o `/dev/null` no Unix. O tamanho do buffer (PINDEX *len*, o segundo parâmetro do *Write()*) é dado em bytes. Nós primeiramente (arquivo `wavchan.cxx`, linha 85) configuramos a variável membro de channel *lastWriteCount* para o número de (sempre com sucesso) bytes escritos. Após isso, nós invocamos o sleep adaptivo ao chamar:

```
writeDelay.Delay(len/2/8);
```

- O objeto *writeDelay* é uma instância da classe PAdaptiveDelay (veja a seção 6.1.3 acima). O método *Delay()* espera receber a duração do sleep em milisegundos. Para obter o número de milisegundos, nós simplesmente dividimos o *len* (que é o tamanho do buffer em bytes) por 2 porque cada amostra ocupa 2 bytes (16 bits) e então dividimos por 8

porque são exatamente 8 amostras a cada um milisegundo (a taxa de amostragem é 8000 Hz).

- O último passo dentro do *Write()* é retornar true para notificar ao chamador que o buffer inteiro foi processado com sucesso (novamente, veja a seção 6.1.3 acima).

6.1.5 Read()

- Vamos agora lidar com o método *Read()* do WavChannel (linhas 93–117). Sua tarefa é ler os dados de áudio de um arquivo WAV.
- O código das linhas 95 até a 102 garante que o *channel* trabalha bem com envio da media mais cedo, quando os canais lógicos são iniciados, antes que o endpoint chamado envie o CONNECT. No caso disto acontecer será enviado uns poucos segundos do arquivo WAV, e talvez a outra parte não esteja preparada para ouvir. Para evitar isso, faz-se a checagem (linha 95) se a conexão H.323 foi estabelecida e se ela não tiver sido, será então preenchido o buffer com silêncio (bytes em zero) ao invés de preencher com os dados do arquivo propriamente dito. Nós naturalmente temos que fazer alguns passos necessários, ex. configurar *lastReadCount* (linha 99) e tomar cuidado do processo de temporização (linha 100). Retornaremos deste método o valor true na linha 101 — a parte restante deste método somente poderá ser executada quando a conexão estiver estabelecida.
- Agora iremos ler um arquivo de áudio especificado em arquivo na linha 104 e se esta leitura falhar, retornaremos o valor false imediatamente. Se a operação de leitura do arquivo tiver sucesso, nós configuramos o *lastReadCount* do channel com o valor obtido do método *LastReadCount()* do wavfile e então (linha 108) chamaremos o sleep adaptativo (*lastReadCount/2/8* que avalia o número de milisegundos que correspondem ao número de amostras lidas do arquivo, veja também 6.1.3 e 6.1.4).
- As linhas 110–114 cuidam da situação quando a operação de leitura retorna menos dados do que a quantidade necessária (ex. *len*). Espera-se que isso aconteça quando tivermos alcançado o final do arquivo de áudio. Então nós vamos desconectar a chamada H.323 neste momento, deste modo devemos chamar o *myConnection.ClearCall()* na linha 113. Observe que o *ClearCall()* não faz a desconexão completa da chamada. Ele apenas inicia a finalização da chamada e retorna, de modo que nosso método *Read()* terá tempo para executar até o final. As ações de finalização de chamada são realizadas em paralelo por outra thread.
- O método termina na linha 116 com uma linha de código que retornará true se pelo menos um byte tiver sido lido, em conformidade com o requisitado em `pwlib/include/ptlib/channel.h` (veja também 6.1.3).

6.2 NullChannel

- A classe NullChannel tem o intenção de se comportat como o `/dev/null` do Unix para escrita e como o `/dev/zero` para leitura. O fonte da classe

é relativamente simples e reusa algum código de WavChannel, deste modo nós não a descreveremos em detalhes.

- O método *Write()* de NullChannel é o mesmo do WavChannel::*Write()*. Ele ignora todos os dados passados por ele, mas relata que estes foram escritos com sucesso — veja 6.1.4 acima. O método *Read()* de NullChannel preenche o buffer passado para ele com silêncio (bytes zeros). E tanto *Read()* quanto *Write()* usam o sleep adaptativo — veja as referências em 6.1.3 até 6.1.5.

Apêndice - Arquivos Fonte da Aplicação Tutorial

- ep.cxx
- ep.h
- Makefile
- nullchan.cxx
- nullchan.h
- oh323tut.cxx
- oh323tut.h
- pconf.h
- wavchan.cxx
- wavchan.h
- **Faça o Download do Código-Fonte Completo deste Tutorial**
- **Versão Antiga do Tutorial**