UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CODEC DE AUDIO CON PÉRDIDA DE PAQUETES PARA TELÉFONOS MÓVILES

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

ALEJANDRO ANDRÉS OPAZO CABAÑA

PROFESOR GUÍA:
PATRICIO NELLEF INOSTROZA FAJARDIN

MIEMBROS DE LA COMISIÓN:
JOSE MIGUEL PIQUER GARDNER
JAVIER ALEJANDRO BUSTOS JIMENEZ

SANTIAGO DE CHILE
AGOSTO 2008

RESUMEN DE LA MEMORIA PARA OPTAR AL
TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: ALEJANDRO OPAZO CABAÑA

“CODEC DE AUDIO CON PÉRDIDA DE PAQUETES PARA TELÉFONOS MÓVILES”

En este trabajo de título se diseña y elabora un programa para ejecutarse en un teléfono celular, que permite junto con simular la trama completa de una llamada VoIP, evaluar distintos tipos de reconstrucciones para la voz en caso de sufrir pérdida de paquetes IP.

Hoy en día, la penetración que tienen los teléfonos celulares y las redes de datos como Internet, han dado paso a un sinnúmero de posibilidades y aplicaciones que hacen uso combinado de estas tecnologías. El envío de audio utilizando dispositivos móviles y paquetes IP, representa todo un desafío pues es considerado una tecnología emergente y se espera que en el futuro tenga un gran impacto dada la transición a la tecnología 3G que recién se está iniciando en Chile.

La transmisión de datos sobre Internet sufre de pérdidas de paquetes, lo que significa que hay un porcentaje de los elementos de comunicación que no llega a su destino. El envío de voz y audio no está excento de este problema y la documentación sobre cómo solucionar este problema en dispositivos móviles aún es pobre.

Se estudió a fondo el comportamiento de la voz tal como se procesa en una llamada telefónica, utilizando el mismo codec AMR. Con esto se diseñó una aplicación de simulación para toda la trama que sufre una onda de audio, desde que es emitida por una persona hasta que llega a su receptor, pasando justamente por un simulador de pérdidas de paquetes. Finalmente se diseñaron y evaluaron distintos tipo de reconstrucciones para suplir las pérdidas sufridas.

En base a las evaluaciones realizadas se obtuvo un prototipo de codec que soporta la pérdida de paquetes de calidad aceptable cuando éstas no superan el 30%. Además la aplicación lograda permite la implementación y evaluación de nuevos prototipos.

Se proponen mejoras para esta aplicación con respecto a la optimización de estructuras de datos y memoria, así como también posibles integraciones con otras aplicaciones ya existentes.

Quisiera agradecer a mi familia, en especial a mis padres por su apoyo incondicional durante estos largos, interrumpidos y difíciles años de estudio. A mi hermana por servirme de inspiración en cuanto al esfuerzo y la dedicación con la que se deben enfrentar los proyectos y a mi cuñado por la fe y el cariño que muestra por mi.

También a Patricio Inostroza y al equipo de NicLabs por haber creido y confiado en este proyecto.

A mis compañeros Claudio Basoalto, Daniel Retamales y Alvaro Neira, pues fueron más que un aporte durante las clases, después de ellas o en cualquier otro ambiente. Compañeras casi no tuve, pero siempre hubo alguna que otra industralilla sentada por ahí que me motivó para venir a la u y entrar a clases.

A mis amigos Fede Nacif, Sebastián Urrutia y Javier Piana, simplemente por estar ahí.

A los cabros del pool, porque aunque los vicios cambiaron a cartas, partidos de la champions, asados, visitas a la terraza y otras deformaciones del terror, terminé pasando varios ramos ahí.

A mis compañías de teatro musical, en especial a Tea-m-danz, por dejar que faltara a mil ensayos para poder dedicarme a esta memoria :) y a mi bandas Kañón y Zhivagos, pues la música es la que me da fuerzas para seguir adelante cuando ya no puedo más.

Y finalmente agradecer a mis musas, que por supuesto son parte de la pasión e inspiración que da paso a la creación tanto de una línea de código como de una obra musical completa y llena de armonías.

Índice de Contenidos

Índice de Figuras

Índice de Tablas

CAPÍTULO 1: INTRODUCCIÓN

1.1 La Telefonía Celular

En sus inicios, finales de los 70, la telefonía celular fue concebida estrictamente para la transmisión de voz, sin embargo, poco a poco la tecnología involucrada en ellos ha ido creciendo, hasta convertirse en una herramienta primordial para la gente común y de negocios.[1]

Hoy en día, los dispositivos móviles tienen gran relevancia en lo que a comunicaciones entre personas se refiere pues cuentan con un sin número de aplicaciones que facilitan la vida: relojes, agendas, juegos, grabadoras de voz, mapas, editores de texto, visores, clientes de correo, navegadores, administradores de archivos, reproductores de mp3 y video, cámaras fotográficas e incluso linternas, que millones de personas utilizan para como medio de contacto o de información, facilitándose la vida y accediendo a canales de información de manera casi inmediata.

Con todas estas herramientas disponibles, no cabe duda que los celulares han cambiado los hábitos de la gente, desde su aparición hace ya más de 20 años, hasta transformarse hoy en día en un aparato imprescindible para muchas personas.[1] En Chile, la penetración de la telefonía celular en la población es de un 74%, cifra con la que es líder en Sudamérica seguido de Argentina con un 68%.[2]

1.2 La Telefonía IP

La penetración que tiene Internet actualmente supera cualquier expectativa realizada al momento de su aparición, asombrándonos con incrementos impensados en lo que a cantidad de sitios se refiere y experimentando cambios significativos respecto a la participación y a los contenidos a los que los usuarios hoy pueden acceder.

El servicio de telefonía vía IP, más conocido como VoIP, utiliza las redes de datos, como Internet, para transmitir la voz. VoIP, acrónimo para "Voice over Internet Protocol" (voz sobre el protocolo Internet), comenzó el año 1995 como resultado del trabajo de un grupo de jóvenes en Israel para comunicar voz entre dos computadores. Para el año 2000 ya se contaba con software y hardware que permitía realizar comunicaciones tanto de PC a Teléfono, como vicecersa.[3] Hoy, el envío de audio vía redes de datos es considerado una tecnología emergente [4] y VoIP un conjunto de recursos utilizados para transmitir voz utilizando redes de datos, específicamente en Internet [5] y entre las principales ventajas que ofrece este servicio, se tiene que:

- VoIP cuesta muchísimo menos que una llamada hecha a través de una compañía telefónica convencional o PSTN. Cifras indican que se puede llegar a ahorrar hasta un 80% de los costos en telefonía.[6]
- No interfiere mayormente en el desempeño de una red, pues el consumo de ancho de banda es marginal, aproximadamente 30kb dependiendo del tipo de comunicación.[6]
- Permite tener un número fijo en el cual recibir las llamadas. Para esto sólo se debe poder acceder a Internet.[7]

Desafortunadamente, el tema de la voz sobre IP para dispositivos móviles está muy pobremente documentado y según lo anterior todo parece indicar que asociado a las nuevas redes IMS[i] en desarrollo, lo próximo en telefonía móvil será justamente celulares con transmisión de voz sobre IP.

1.3 Problemas Asociados a la Transmisión de Datos

La transmisión de datos sobre Internet sufre de pérdida de paquetes, lo que significa que hay un porcentaje de los elementos de comunicación (paquetes de datos), que luego de viajar a través de la red, falla al alcanzar su destino.[8] En un sistema de comunicación por paquetes, la velocidad de la computadora del usuario final, el ancho de banda, servidores Web saturados, enrutamiento en la red de baja calidad (debido al exceso de saltos o a las distancias muy grandes) o congestión en la red [9] producen una degradación de la calidad de la señal recibida. La transmisión de voz, no es una excepción a este problema, siendo este aún más crítico pues los

i Ver descripción de IMS en página 12

paquetes retardados ya no son útiles para ser reproducibles, aunque pueden ser usados para elaborar modelos de reconstrucción para otros paquetes perdidos. [10]

La degradación en la señal, genera 3 problemas bien conocidos en VoIP [11]:

- **Latencia:** se define técnicamente en VoIP como el tiempo que tarda un paquete en llegar desde la fuente al destino y se da por el procesamiento que enfrenta la señal tanto en el emisor como en el receptor. No hay una solución que se pueda implementar de manera sencilla. Se puede intentar reservar un ancho de banda de origen a destino o señalizar los paquetes con valores de TOS (Tipo de servicio: suele corresponder a un campo de 8 bits de la cabecera de los datagramas IP que identifica la prioridad relativa de un paquete sobre otro) para lograr que los equipos sepan que se trata de tráfico en tiempo real y lo traten con mayor prioridad, pero actualmente no suelen ser medidas muy eficaces ya que no disponemos del control de la red.

  La latencia o retardo entre el punto inicial y final de la comunicación debiera ser inferior a 150 ms. El oído humano es capaz de detectar latencias de unos 250 ms, 200 ms en el caso de personas bastante sensibles. Si se supera ese umbral la comunicación se vuelve molesta.
- **Jitter:** es un efecto de las redes de datos no orientadas a conexión y basadas en conmutación de paquetes. Como la información se discretiza en paquetes, cada uno de los paquetes puede seguir una ruta distinta para llegar al destino, lo que puede resultar en que los paquetes lleguen a destino en un orden completamente distinto al original. La solución más ampliamente adoptada es la utilización del jitter buffer[ii], que consiste básicamente en asignar una pequeña cola o almacén para ir recibiendo los paquetes y sirviéndolos con un pequeño retraso. Si algún paquete no está en el buffer cuando se necesita (se perdió o no ha llegado todavía), entonces se descarta. Normalmente se pueden modificar los buffers: un aumento del buffer implica menos pérdida de paquetes pero más retraso, una disminución implica un menor retardo pero una mayor pérdida de paquetes.

  El jitter entre el punto inicial y final de la comunicación debiera ser ser inferior a 100 ms. Si el valor es menor a 100 ms el jitter puede ser compensado de manera apropiada. En caso contrario debiera ser minimizado.

ii La palabra buffer, se traduce como contenedor, sin embargo se usará la palabra en inglés debido a que es ampliamente usada en el ámbito de la computación y audio. También se utilizará buffers como el plural de buffer.

- **Eco** o **Reverberación:** La latencia y el jitter pueden producir eco sobre el sistema telefónico. Pero también puede producirse por un retorno de la señal que se escucha por los altavoces y se cuela de nuevo por el micrófono. El eco se define como una reflexión retardada de la señal original y puede ser tratado mediante *Supresores* (evitando que la señal se devuelva) o *Canceladores* (guardando la información enviada para posteriormente filtarla).

  El oído humano es capaz de detectar el eco cuando su retardo con la señal original es igual o superior a 10 ms. Pero otro factor importante es la intensidad del eco ya que normalmente la señal de vuelta tiene menor potencia que la original. Es tolerable que llegue a 65 ms y una atenuación de 25 a 30 dB.

1.4 Medición de la calidad de una llamada VoIP

Hay muchos métodos para medir la calidad de una llamada de VoIP, pero en general existen dos grandes grupos: de forma intrusiva o con tráfico real. [12]

La forma intrusiva implica equipos en ambos extremos de la comunicación y no se realiza en tiempo real. Las pruebas consisten en enviar una señal conocida a través de la red, capturarla en el otro extremo y compararla con la señal enviada. Debido a la dificultad de analizar dos señales, los equipos que utilizan este método tienen una complejidad elevada y no pueden realizar el análisis en tiempo real. Los algoritmos más utilizados para esta comparación son:

- **PSQM** (Perceptual Speech Quality Measurement): Las medidas se realizan comparando la señal original con la señal que se ha recibido codificada en el otro extremo de la comunicación, así se obtiene un valor PSQM. Sin embargo, esta medida sólo considera los efectos de la compresión y/o descompresión del códec y no tiene capacidad de analizar los efectos causados por el trayecto a través de la red, como pueden ser la pérdida o el jitter de paquete.
- **PESQ** y **PAMS**, fueron diseñados para aumentar el rango que cubrían las medidas PSQM, incluyendo distorsión, filtrado y otras desigualdades del canal. Pero tampoco analizan todos los factores.

El grupo de análisis pasivo, utiliza un equipo analizador en un solo extremo de la comunicación y puede ser realizado con tráfico real (y en tiempo real), pues analizan la calidad de la voz sin interferir en las llamadas existentes y sin necesidad de una señal de referencia. Algunos ejemplos son [12]:

- **E-Model** (valor R), proporciona un solo valor llamado R que se deriva de las características de la red, como el retardo y otros valores. El éxito de este análisis es que proporciona el valor MOS sin utilizar a toda la gente necesaria para el experimento estadístico, aportando el valor de una forma exacta. El valor de R varía entre 0 (muy poca calidad) a 100 (muy alta calidad). Cualquier valor por encima de 50 es aceptable.
- **VQmon** (Voice Quality Monitoring) deriva del E-Model y analiza los errores de la red (jitter, pérdidas y retardo) al tiempo que predice el impacto en la señal de audio reconstruida. VQmon no considera directamente los aspectos de la conversión analógica-digital.
- **MOS** (Mean Opinion Score) asigna un valor a la calidad de la llamada en toda la red. El valor de MOS real ha sido determinado en un ejercicio estadístico en el cual, un gran número de personas escucha la misma llamada y la valora de 1 (mala) a 5 (excelente). La medida, por lo tanto, tiene en cuenta tanto al códec como los efectos de la red.

En la tabla 1 se pueden apreciar las características principales de percepción de algunos algoritmos de compresión de voz. Mientras que los codecs de fuente son diseñados específicamente para la voz, los codecs de audio funcionan bien con cualquier tipo de sonido. Cabe decir que un R-Rated Factor sobre 90 es equivalente a MOS-4.3 o más y que se considera una calidad aceptable sobre R-80 o MOS-4.0. [13]

CODEC	Bit Rate	Tamaño de un frame de voz en milisegundos	MOS-Mean Opinion Score	R-Rating Factor
CODEC de onda Para todo tipo de audio				
G.711	64 KBPS	10-30 mseg	4.2	80-90 la mayoría satisfechos
G-726	16-40 KBPS	10-30 mseg	3.7-3.9	70-80 algunos usuarios no satistechos
iLPBC	13-15 KBPS	20-30 mseg	3.9	80-90 la mayoría satifechos
CODEC de fuente Preferidos para la voz				
G.729	8 KBPS	10-30 mseg	4.0	80-90 la mayoría satifechos
G.723.1	5.3-6.3 KBPS	30-90 mseg	3.7-3.9	70-80 algunos usuarios no satistechos
GSM-EFR-Enhanced Full Rate using ACELP	12.2 KBPS	20 mseg	4.4	90-100 muy satisfechos

Tabla 1: Algoritmos de compresión de voz

Por todo lo anterior es que se ha determinado estudiar el comporamiento de la voz para telefonía celular, junto con diseñar y evaluar un prototipo de Codec de audio que soporte la pérdida de paquetes.

1.5 Conceptos Básicos

- **Voz sobre Protocolo de Internet**, también llamado Voz sobre IP, VoIP, Telefonía IP, Telefonía por Internet, Telefonía Broadband o Voz sobre Broadband, corresponde al enrutamiento de conversaciones de voz sobre Internet o a través de alguna otra red basada en IP. [14]
- **Session Initiation Protocol** (SIP o Protocolo de Inicialización de Sesiones), es un protocolo desarrollado por el IETF MMUSIC Working Group para crear, modificar y terminar sesiones con uno o más participantes. Una de sus ventajas es que los mensajes SIP son legibles por los humanos, pues reutiliza muchos de los campos de encabezado, reglas de cofidificación, códigos de error y método de autenticación del protocolo HTTP,

lo que además, lo hace menos complejo de entender que H.323 e independiente del protocolo de transporte (puede usar UDP, TCP, ATM, etc). [15]

- **Codec,** es una abreviatura de **Co**dificador-**Dec**odificador. Describe una especificación implementada en software, hardware o una combinación de ambos, capaz de transformar un archivo en un flujo de datos (stream) o una señal.[16]
- **3G,** es una abreviatura para tercera-generación de telefonía móvil. Los servicios asociados con la tercera generación proporcionan la posibilidad para transferir tanto voz y datos (una llamada telefónica) y datos no-voz (como la descarga de programas, intercambio de correos, y mensajería instantánea). [17]
- **H.323**, es un estándar de la ITU (Unión Internacional de Telecomunicaciones) que define una gran cantidad de información acerca de las propiedades y componentes que interactúan en la comunicación multimedia en redes de área local (LAN). Fue diseñado desde el principio para incluir Voz sobre IP y telefonía sobre IP y describe una serie de estándares y protocolos, codificadores permitidos de audio y video, RAS (registro, admisión y estado), señalización de llamadas y señalización de control. Además H.323 define un nivel obligatorio de cumplimiento y soporte de las especificaciones antes mencionadas para todas las terminales en la red. Las debilidades de H.323 apuntan a su complejidad pues su especificación tiene más de 700 páginas y define más de 100 elementos (encabezados) y al hecho de que usa una representación binaria para los mensajes, basado en ASN.1 y en reglas de codificación de paquetes (PER). [18]
- **IMS** (IP Multimedia Subsystem), es un marco arquitectónico, diseñado originalmente por los estándares wireless de 3GPP, para entregar servicios multimedia IP a los usuarios. [19]
- **RTP** son las siglas de Real-time Transport Protocol (Protocolo de Transporte de Tiempo Real). Es un protocolo de nivel de aplicación (no de nivel de transporte, como su nombre podría hacer pensar) utilizado para la transmisión de información en tiempo real, como por ejemplo audio y video en una video-conferencia. [20]
- **UDP** (User Datagram Protocol) es un protocolo del nivel de transporte basado en el intercambio de datagramas. Permite el envío de datagramas a través de la red sin que se haya establecido previamente una conexión, ya que el propio datagrama incorpora suficiente información de direccionamiento en su cabecera. [21]

- **VAD** (Voice Activity Detection) detecta si el audio que está siendo codificado es un discurso, un silencio o un ruido de fondo.
- **DTX** (Discontinuous Transmission) es un complemento de la acción de VAD, que permite detener la transmisión completamente si el sonido de fondo se vuelve estable.
- **AMR** (Multi-tasa adaptativo del inglés Adaptive Multi-Rate) es un formato de compresión de audio optimizado para la codificación de voz. AMR ha sido adoptado como el estándar de codificación de audio por 3gpp en octubre de 1998 y actualmente se utiliza ampliamente en GSM. Gestiona dinámicamente el ancho de banda seleccionando entre ocho diferentes tasas de bits. Los anchos de banda 12.2, 10.2, 7.95, 7.40, 6.70, 5.90, 5.15 y 4.75 kb/s se basan en frames que contienen 160 muestras y que dxuran 20 milisegundos. [22]
- **PCM** es una representación digital de una señal análoga donde la magnitud de la señal es sampleada regularmente a intervalos uniformes y luego cuantizada a una serie de símbolos de un código numérico generalmente binario. PCM se utiliza en sistemas de telefonía digital, en los teclados musicales electrónicos de la era del '80 y es también, un estándar en el audio digital de computadores y de discos compactos (según el libro rojo). A menudo, la codificación PCM facilita la transmisión digital desde un punto a otro en forma serial. [23]

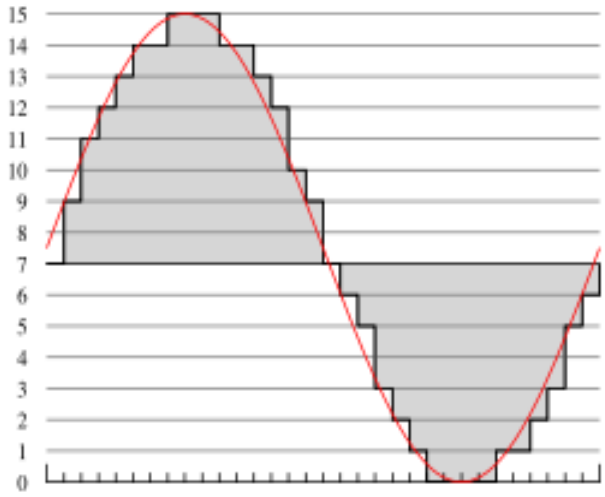

Figura 1: Representación de una señal (línea curva) para PCM de 4 bits

1.6 Alcances

Por ser un tema muy amplio se ha optado por no incluir en este proyecto pruebas del codec en un ambiente real y sólo trabajar mediante la simulación de envío y recepción, para no lidiar con pérdidas no deseadas durante la evaluación del codec. Tampoco se pretende obtener un producto optimizado pues se trabajará en un prototipo.

CAPÍTULO 2: ESTADO DEL ARTE

2.1 Estado del Arte en Soluciones VoIP

Hoy en día, existen variadas soluciones que implementan VoIP a nivel personal y empresarial. Generalmente, las llamadas entre aplicaciones y/o teléfonos IP son gratuitas puesto que sólo se debe contar con conexión a Internet (lo que se considera un costo hundido del servicio, para toda la gente que ya tiene acesso a Internet) y las que son de Teléfono a PC o bien de PC a Teléfono, son de pago pues se debe cancelar, entre otros, el costo de codificar la señal para que sean enviadas a la red de telefonía pública o PSTN (Public Switched Network) tomando la voz y convirtiéndola en paquetes que pueden viajar por Internet y viceversa [24]. Ejemplos de soluciones dedicadas a empresas son: CALL-IP, voipGATE, Eurocomm Group, Pibix, TalkFree, etc. Por su parte, algunas soluciones dedicadas al usuario final son: Asterisk, Ekiga, Jajah, OpenWengo, Skype, Voipbuster y WengoPhone NG, destacando Skype por la rapidez con la que ha incorporado usuarios en los últimos 3 años (hoy cuenta con casi 200.000.000 de usuarios) [25] y por la calidad de servicio que ofrece. Todas estas soluciones no serían, sin embargo, posibles sin la intervención de los codecs, que son los encargados de convertir las señales de audio en datos y viceversa.

2.2 Estado del Arte de los Codecs

Los codecs, cuyo nombre proviene de la combinación de Codificador y Decodificador [26], pueden ser bien definidos por 4 características:

- **Número de canales**: un flujo de datos codificado puede contener una o más señales de audio simultáneamente. De manera que puede tratarse de audiciones "mono" (un canal), "estéreo" (dos canales, lo más habitual) o multicanal. Los códec de audio multicanal se suelen utilizar en sistemas de entretenimiento "cine en casa" ofreciendo seis (5.1) u ocho (7.1) canales.
- **Frecuencia de muestreo**: determina la calidad percibida a través de la máxima frecuencia que es capaz de codificar, que es precisamente la mitad de la frecuencia de muestreo.

- **Número de bits por muestra**: Determina la precisión con la que se reproduce la señal original y el rango dinámico de la misma. El más común es 16 bits.
- **Pérdida**: Algunos códecs pueden eliminar frecuencias de la señal original que, teóricamente, son inaudibles para el ser humano. De esta manera se puede reducir la frecuencia de muestreo. En este caso se dice que es un *códec con pérdida* o *lossy codec* (en inglés). En caso contrario se dice que es un *códec sin pérdida* o *lossless codec* (en inglés).

En videoconferencia existen estándares definidos dentro del conjunto de normas ITU H.320 y H.323, tales como: G.711, G.722, G.723, G.728, G.729. Diversos códecs admiten diversas velocidades para adecuarse a la capacidad de transmisión de las redes de comunicaciones subyacentes. En cuanto a codecs de audio *con pérdida*, existe una gran variedad que responde a las muy diversas necesidades que se ha intentado satisfacer. Algunos ejemplos son: MP1, MP2 y MP3 (correspondientes a MPEG1 audio layer-1, 2 y 3, respectivamente), Advanced Audio Coding (AAC), Ogg Vorbis, WMA (Windows Media Audio), MPC (Musepack), AC3 (Dolby Digital A/52), DTS (Digital Theather Systems), ADPCM, ADX (usado en videojuegos), ATRAC (Adaptive TRansform Acoustic Coding), Perceptual Audio Coding y TwinVQ. Existen además codecs de audio que han sido optimizados para transmitir voz, por la sencilla razón que el audio involucrado no requiere de una mayor capacidad para reflejar toda su riqueza y complejidad, éstos son: AMBE, AMR, CELP, EVRC, G.711 (Ley Mu y Ley A), G.722, G.723, G.726, G.728, G.729, GSM, HILN (MPEG-4 Parametric audio coding), iLBC, IMBE, Perceptual Audio Coding (usado en radio digital y vía satélite), Speex (libre de patentes), SMV, QCELP, VSELP [27].

En la tabla 2, podemos ver un cuadro que compara características entre los codecs de voz.

Codec	Rate (kHz)	BitRate (kbps)	Delay frame + lookahead (ms)	Multi-rate	Embedded	VBR	PLC	Bit-robust	Licencia
Speex	8, 16, 32	2.15-24.6 (NB) 4-44.2 (WB)	20+10 (NB) 20+14 (WB)	Si	Si	Si	Si		Open-source/ software libre
iLBC	8	15.2 or 13.3	20+5 or 30+10				Si		Gratis, pero de código cerrado
AMR-NB	8	4.75-12.2	20+5	Si			Si	Si	Propietario
AMR-WB (G.722.2)	16	6.6-23.85	20+5	Si			Si	Si	Propietario
G.729	8	8	10+5				Si	Si	Propietario
GSM-FR	8	13							Patentado
GSM-EFR	8	12.2					Si	Si	Propietario
G.723.1	8	5.3 6.3	37.5				Si		Propietario
G.728	8	16	0.625						Propietario
G.722	16	48 56 64			Si				

Tabla 2: Características de los codecs de voz

La información contenido en la Tabla 2 se explica a continuación:

- **Codec**: indica el nombre del codec
- **Rate**: indica la frecuencia de muestreo de la señal vocal, es decir, cada cuánto se toma una muestra de la señal analógica
- **BitRate**: indica la cantidad de información que se manda por segundo.
- **Delay frame + lookahead**: indica el retardo que producen: el envío mismo del paquete y el hecho de tener que consultar paquetes extras para reconstruir la información sonora.
- **Multi-rate**: permite al codec cambiar de bitrate dinámicamente en cualquier momento
- **Embedded**: si el codec puede enviar *narrowband bitstreams* en *wideband bitstreams* sin ocupar recursos extras.
- **VBR**: acrónimo para Variable BitRate (Bitrate Variable)

- **PLC**: acrónimo para Packet Loss Concealment (tratamiento de los paquetes perdidos)
- **Bit-robust**: señala si el codec es robusto a la corrupción a nivel de bits como algunos usados en redes inalámbricas.
- **Licencia**: tipo de licencia bajo la que se distribuye el codec

En la Tabla 3 podemos ver un cuadro comparativo sobre los detalles técnicos de varios códecs de audio. Los conceptos recién explicados en la tabla 2 de Codec, Rate, BitRate y VBR son también aplicables a la tabla 3. Los demás términos utilizados en el encabezado de cada columna, se explican a continuación:

- **Algoritmo**: indica si los codecs son codecs con pérdida, sin pérdida, híbridos u orientados a las voz.
- **Bits per sample**: indica el número de bits que tiene cada muestra sonora.
- **CBR**: acrónimo para Constant Bitrate (Bitrate Constante).
- **Stereo**: indica si el codec tiene la capacidad de codificar señales estéreo.
- **Mono**: indica si el codec tiene la capacidad de codificar señales mono.

Codec	Algoritmo	Rate	Bit rate	Bits per sample	CBR	VBR	Stereo	Mono
AAC	Lossy, Hybrid	8 Hz to 192 kHz	8 to 529 kbit/s (stereo)	Cualquiera (típicamente usa el fp interno)	Si	Si	Si: Dual, Mid/ Side, Intensity, Parametric	Si
MP3	Lossy	8, 11.025, 12, 16, 22.05, 24, 32, 44.1, 48 kHz	8, 16, 24, 32, 40, 48, 56, 64, 80, 96, 112, 128, 160, 192, 224, 256, 320 kbit/s	Cualquiera (típicamente usa el fp interno)	Si	Si	Si: Dual, Mid/ Side, Intensity	Si
RealAudio	Lossless, Lossy	Varios	Varios	Varios	Si	Si	Si	Si
Speex	Speech	8, 16, 32 (48) kHz	NB: 2.15 to 24.6 kbit/s WB: 4 to 44.2 kbit/s		Si	Si	Si: Intensity	Si
Vorbis	Lossy	1 Hz to 200 kHz	Variable	Cualquiera (típicamente usa el fp interno)	Si	Si	Si: Dual, Lossless, Phase, Point (Intensity)	Si
WavPack	Lossless, Lossy, Hybrid	1 Hz to 16777.216 kHz	Variable modo sin pérdida; sobre 196 kbit/s en modo con pérdida (para CD de audio)	Varios en modo sin pérdida; 2.2 mínimo en modo sin pérdida	Si	Si	Si	Si
Windows Media Audio	Lossless, Lossy	8, 11.025, 16, 22.05, 32, 44.1, 48, 88.2, 96 kHz	4 to 768kbit/s, Variable para codificación sin pérdida	16, 24 en modo sin pérdida, cualquiera en modo con pérdida (típicamente usa el fp interno)	Si	Si	Si	Si

Tabla 3: Detalles técnicos de distintos codecs de audio

De todos estos codecs, llama la atención **iLBC** (internet Low Bitrate Codec), pues está diseñado para voz sobre IP por *Global IP* Sound. Con iLBC, los frames son codificados completamente independientes, lo que ofrece una mejor calidad cuando las tasas de pérdida son del 10% o más. Claramente el contra de esto, es que bajo condiciones normales, este codec no es óptimo y mucho menos utilizable en telefonía móvil pues el retardo con tasas de pérdida altas sería intolerable.

2.3 Estado del Arte de Reconstrucción de Señales de Voz

Para enfrentar la pérdida de paquetes, se han desarrollado diferentes técnicas, como:

- **Sustitución por ceros [28]**: Se reemplazan las muestras perdidas por ceros.

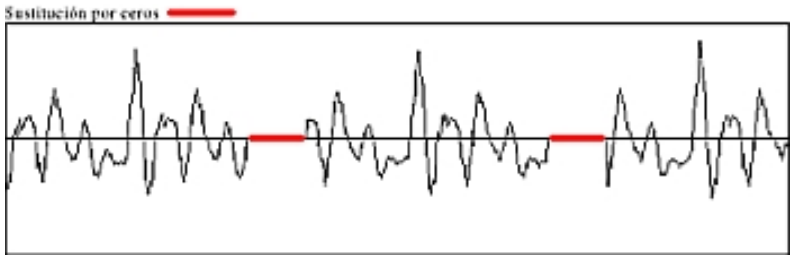

Figura 2: Sustitución por ceros

- **Repetición [29]**: Se repite el último paquete recibido.

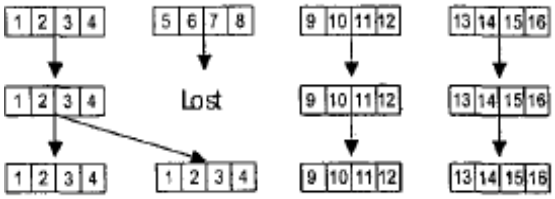

Figura 3: Repetición

- **Repetición por planilla [30]**: Este es un algoritmo simple que toma el paquete inmediatamente anterior al perdido "A" y buscando hacia atrás, selecciona el paquete más

parecido “B”. Luego copia posterior a “A” el patrón que sigue a “B”. A continuación se puede ver un diagrama:

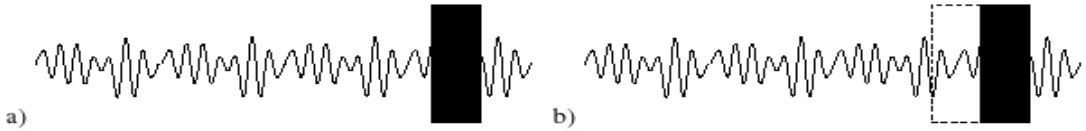

Figura 1: a) el cuadro negro representa el paquete perdido b) el cuadro a trazos representa el paquete inmediatamente anterior.

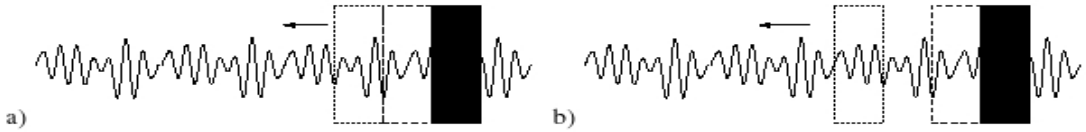

Figura 2: a) y b) representan a la búsqueda del mejor candidato.

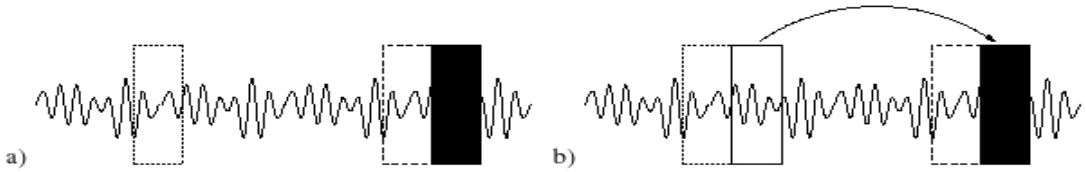

Figura 3: a) Selección del mejor candidato b) Copia del paquete siguiente al lugar correspondiente.

Figura 4: Repetición por planilla

- **Aleatorización de los paquetes [29]**: Esta técnica busca disminuir el efecto de la pérdida de paquetes, distribuyendo la información de un discurso en varios paquetes. Esta técnica debe ser cuidadosamente aplicada, pues la latencia producida al distribuir los paquetes y luego al reorganizarlos en el receptor puede llegar a ser insoportable para una comunicación en tiempo real.

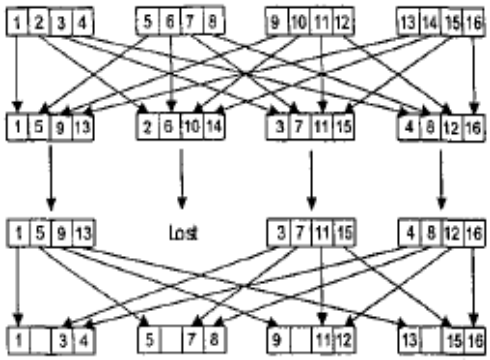

Figura 5: Aleatorización de los paquetes

- **Aleatorización con repetición (VREN) [29]**: La idea es aplicar la técnica de aleatorización antes de enviar la información y la técnica de repetición en el receptor para reparar los paquetes perdidos.

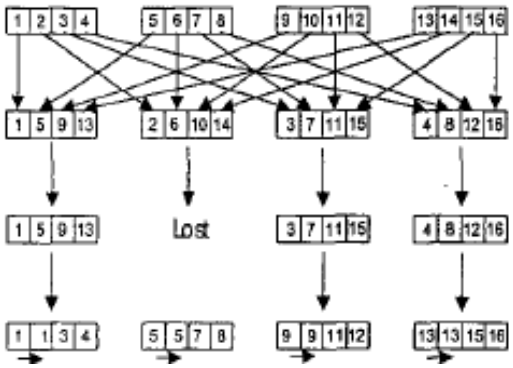

Figura 6: Aleatorización con repetición

- **Aleatorización con interpolación (IEN) [29]**: la idea es combinar la aleatorización en el emisor con la interpolación simple para reparar los paquetes perdidos. La interpolación simple consiste en calcular el promedio entre los paquetes anterior y posterior al paquete perdido. Esta técnica tiene por ventaja su simplicidad, pero la eficiencia se degrada en la medida que el número de paquetes perdidos es mayor.

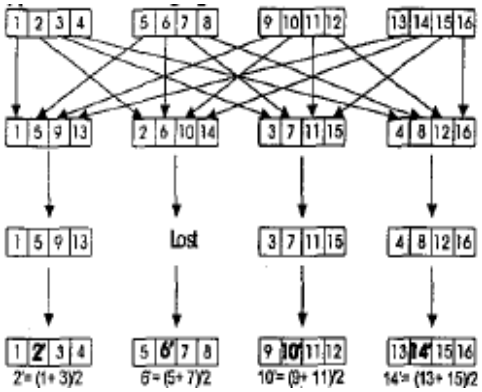

Figura 7: Aleatorización con interpolación

- **XOR [29]**: aplica redundancia de los paquetes *n-1* en el paquete *n*. Luego mediante la aplicación de XOR (ó exclusivo) a los *n-1* paquetes anteriores se puede obtener el paquete

n. Así, si sólo se pierde un paquete entonces, éste es recuperable.

- **Lagrange [29]**: En este método la idea es aplicar el polinomio de Lagrange para recuperar la información perdida usando unidades del texto anterior y posterior a la unidad perdida.

El nivel de reconstrucción de algunas de estas técnicas se puede apreciar la figura 8, con datos obtenidos de un estudio en el que se evaluó el reconocimiento de voz en el idioma francés para distintos escenarios de transmisión y de degradación. [29]

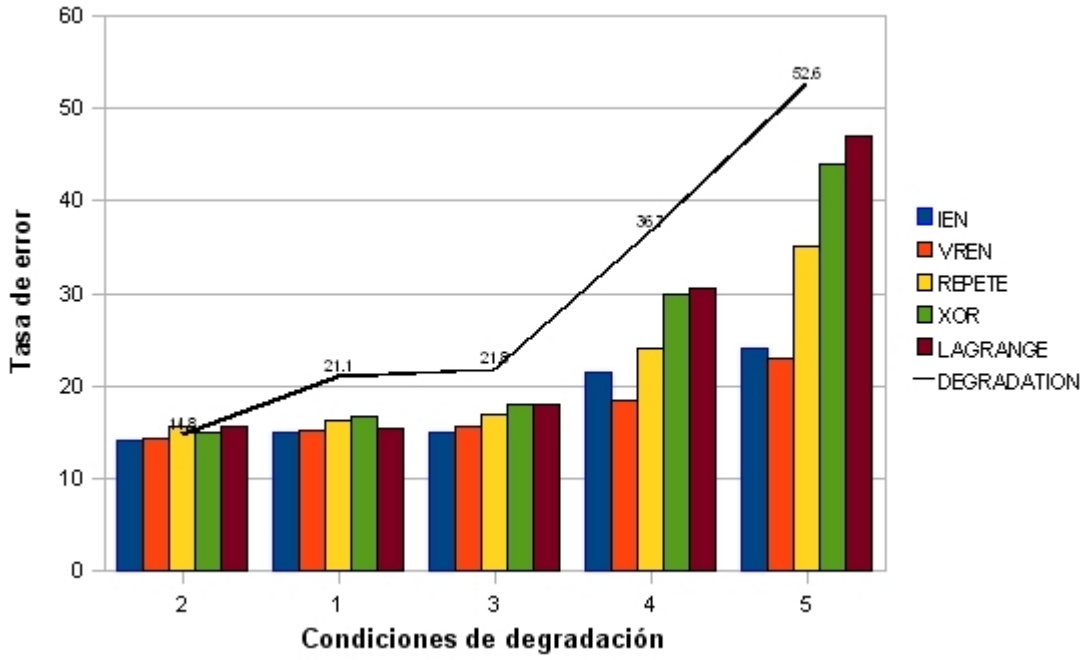

Figura 8: Nivel de reconstrucción de señales de voz

CAPÍTULO 3: SYMBIAN

Symbian es un sistema operativo desarrollado prácticamente por Nokia (actualmente propietario del 40% de acciones) y por eso los moviles Nokia llevan este sistema operativo, aunque también otras marcas como Sony Ericsson o Siemens utilizan este sistema en sus terminales móviles. [31]

De acuerdo a la documentación, Symbian OS es "un sistema operativo grande de cientos de clases y miles de funciones. Parece ser uno de los sistemas más complicados de operar jamás construidos. La jerarquía básica de clases de Symbian contiene 1201 clases. El kit de desarrollo para la serie 60 agrega 293 clases y el kit para UIQ añade 705 clases más! Hay muchas clases con cientos de métodos en su interfaz, y la mayoría de ellas tiene al menos 10 métodos. Ningún ser humano puede comprender en su totalidad un tipo de sistema como éste". [32]

Para comprender el trabajo realizado, así como sus beneficios y alcances, es necesario conocer algunos aspectos fundamentales del lenguaje de programación nativo en este sistema llamado Symbian C++, tema del cual se trata el presente capítulo.

3.1 Symbian C++

El kernel de Symbian OS forma un compacto sistema operativo multitarea, con muy poca dependencia de los periféricos. Los accesos al hardware únicamente pueden ser realizados en modo privilegiado. Las aplicaciones de Symbian OS trabajan siempre en modo usuario, por lo que no pueden acceder directamente al hardware. El kernel pone los servicios del hardware a disposición de las aplicaciones a través de APIs. Cuando las aplicaciones desean interactuar con el hardware del dispositivo tienen que hacer una petición a través de una interfaz, de modo que éstas se comportan como clientes, mientras que las APIs del sistema operativo actúan como servidores. Symbian OS ha sido diseñado bajo este patrón *cliente-servidor* y en el uso de eventos.

Para entender una aplicación hecha en Symbian C++, se debe tener en cuenta que éstas

siempre siguen el patrón MVC *(modelo-vista-controlador)*. Para desarrollar entonces una aplicación, se deben escribir al menos 3 clases para satisfacer a este patrón (en la práctica la mayoría de los proyectos no tienen menos de 20 archivos). El primer paso entonces es escribir la clase del modelo, que usualmente tiene el sufijo de `Document`. El segundo paso, consiste en proveer una clase controladora, la cual es, desconcertantemente llamada `AppUi`. Lo extraño del nombre acá es que esta clase no muestra directamente en pantalla ninguna interfaz para el usuario, en cambio, contiene los manejadores de eventos generados tanto por la aplicación como por el usuario. Finalmente, los elementos de la interfaz para el usuario (la parte *vista* del patrón) están definidos en las clases que usualmente son nombradas con el sufijo `Container` ó `View`.

Además de las clases correspondientes al patrón MVC, Symbian C++ requiere de una clase `Application`.

La estructura básica de un proyecto de 3a Edición consta de los siguientes directorios y archivos:

- **Src: Archivos fuente C++**
  - .cpp archivos fuente
  - .hrh cabeceras compartidas entre archivos .cpp y .rss. Contiene las deficiones requeridas para usar un recurso desde el código C++
- **Inc: C++ archivos cabecera (inclusiones)**
  - .h archivos de cabecera comunes
- **Group: Archivos de empaquetamiento y armado (build)**
  - .inf y .mmp usados para compilar aplicaciones en el sistema Symbian
  - .pkg instrucciones sobre cómo crear un paquete, es decir, un archivo instalable (.sis)
  - .rss describe una intefaz gráfica
  - .rsc archivos .rss compilados con el compilador de recursos (rcomp, resource compiler)
  - .hrh cabeceras compartidas entre archivos .cpp y .rss. Contiene las deficiones requeridas para usar un recurso desde el código C++
  - .rh archivos .rss pueden incluir a éstos indicando dónde están almacenadas las

cabeceras

- **Aif: Archivos con información para crear el ícono de la aplicación (Application Information File)**
  - .rss describe la interfaz gráfica usando archivos de recursos
  - .rsc archivos .rss compilados con el compilador de recursos (rcomp, resource compiler)

3.2 El concepto Leave

Symbian C++ no usa el mécanismo estándar de manejo de excepciones de C++, pues fue considerado muy intensivo en memoria en el tiempo en que Symbian OS fue diseñado. En reemplazo, existe un mecanismo llamado **Leave**, el cual es una manera liviana de manejar excepciones.

Este mecanismo es usado en todo el sistema Symbian y puede ser reconocido en las funciones que tengan el sufijo L en su nombre, por ejemplo ConstructL. Cuando una función tiene tal nombre, entonces puede hacer leave, en otras palabras, puede retornar un estado especial de error que propagará la necesidad de un retorno inmediato. Cuando una función retorna desde una llamada en un estado de error, retorna inmediatamente la función padre, recursivamente, también en estado de error hasta que alguien la atrape y se haga cargo de ella. Esto ocurre cuando un *trap* es encontrado.

Nota: cuando ocurre un leave, se retorna inmediatamente, por lo que no existe la posibilidad de liberar la memoria reservadas para los objetos en el heap. Éste es un problema grave en la limitada memoria de los dispositivos, por eso se debe asegurar la limpieza de memoria cuando ocurre un leave. La limpieza de memoria está asociada a una pila llamada Cleanup Stack sobre la cual no se hablará en detalle, pero si se asumirá que el lector conoce. [33]

3.3 Contrucción de Dos Fases

Cuando se contruye un objeto con new, a menudo se debe reservar memoria para la

creación de sub-objetos. Si eventualmente ocurriera un leave en el constructor, no se invocaría al destructor y no habría posibilidad de poner los sub-objetos en la pila de limpieza (cleanup stack).

La solución a esto es realizar la construcción en dos fases. En la primera fase, se realiza la construcción en el constructor, pero siguiendo la regla que en el constructor no puede ocurrir un leave. Esto quiere decir que un constructor no puede reservar memoria, porque un eventual falta de este recurso produciría un leave.

La construcción real, con la efectiva reserva de memoria para sub-objetos, es realizada por un método llamado por convención ConstructL que si puede hacer leave. El destructor también debe ser escrito de una manera especial: debe funcionar en ambas situaciones, si el objeto fue creado completamente (bajo ConstructL) y también si no (sólo se hizo la llamada al constructor).

Dado que el llamar a ConstructL después de crear cada objeto es tedioso, usualmente se usan dos métodos estáticos, llamados por convención NewL y NewLC. El primer método o crea un objeto completo o hace Leave, limpiando los sub-objetos intermedios. El segundo método o crea un objeto completo agregándolo a la pila de limpieza o hace Leave.

3.4 Descriptores

La decisión de Symbian de no usar los tipos de datos y funciones comunes para manejar strings y buffers binarios puede parecer sorprendente para muchos desarrollares nuevo en esta plataforma. Probablemente muchos pasaron más de un par de días preguntándose los méritos de TBuf, TBufC, HbufC. Así pues, las características principales de los descriptores en Symbian son:

- strings y datos binarios son tratados de la misma manera.
- los datos pueden residir en cualquier ubicación de memoria, ya sea, ROM o RAM, en el stack o el heap.

- un objeto descriptor mantiene un puntero a la largo de la información que describe sus propios datos. Algunos descriptores también incluyen el largo máximo.

En la figura 9 se muestra la jerarquía de clases de los descriptores:

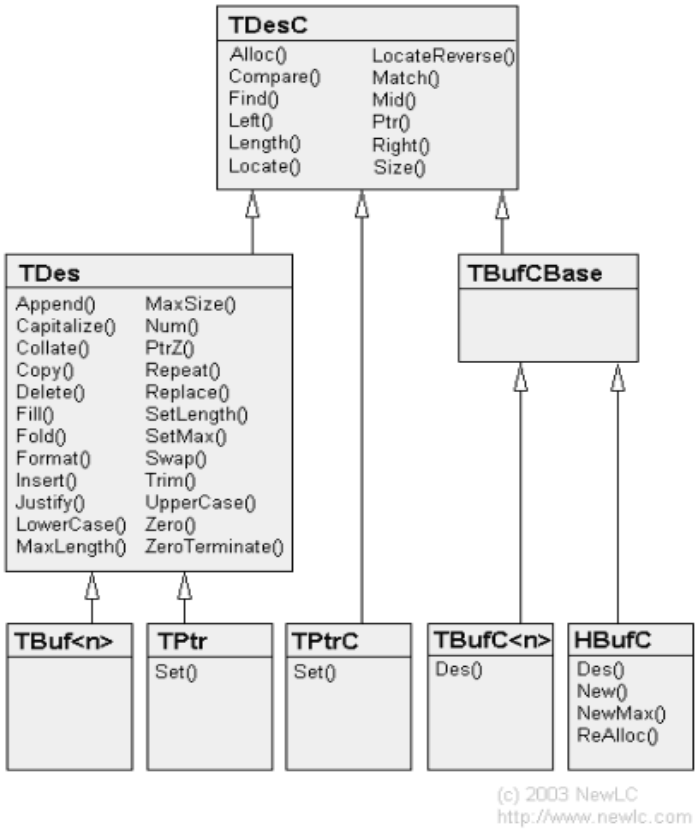

Figura 9: Jerarquía de clases de los descriptores

Todos los descriptores descienden de la clase abstracta `TDesC`. Existen 3 tipos:

- Descriptores Buffer: donde los datos son parte del objeto descriptor y son alojados en el stack. Estos son: `TBuf` y `TbufC`.
- Descriptores Heap: donde los datos son parte del objeto descriptor y son alojados en el heap. Existe uno solo de este tipo: `HBufC`.
- Descriptores Puntero: donde el objeto descriptor está separado de los datos que representa. Estos son: `TPtr` y `TPtrC`.

Análogamente en C/C++ los tipos de datos se podrían ver como:

- `TPtrC` debería ser usado donde se usaba const '`char *`'
- `TBufC` en el lugar de '`char []`'

Las otras clases no tienen equivalencias.

El cómo se organizan los datos en cada clase se puede ver en la figura 10:

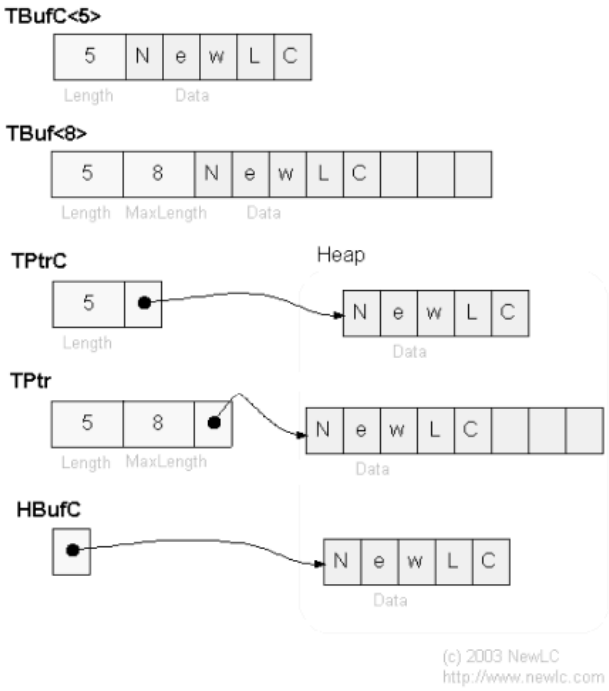

Figura 10: Descriptores en Symbian

`TDes` y `TDesC` son clases abstractas y no pueden ser instanciadas. Se utilizan principalmente como argumento en alguna función que maneje strigns o datos binarios. En tal caso, se debe usar:

- `const TDesC&` para datos o strings que sólo serán leídos (read-only)
- `TDes&` para strings y datos que se desea modificar (read & write)

Todos los descriptores pueden tener un indicador de ancho: `TDes8`, `TDes16`, `TDesC8`, `TDesC16`, `TBuf8`, `Tbuf16`,... El sufijo 8 indica que el descriptor maneja datos en 8 bits, y 16, lo mismo para 16 bits. Generalmente, se usa la forma neutral (`TDes`, `TDesC`,...) para datos de textos y la versión de 8 bits (`TDesc8`,...) para contenido binario.[34]

3.5 Los Objetos Activos

Los Objetos Activos constituyen un paradigma de programación realmente adaptado para estos casos. La facilidad de programar con Objetos Activos radica en que no se tienen que manejar exclusiones mutuas con otros threads cuando se quiere trabajar con los mismos recursos.

Más aún, los Objetos Activos emplean, generalmente, menos recursos que los threads. Un ejemplo: el tamaño de memoria que usan los threads bordea los 4K en el caso de un thread en el kernel y los 12K para un thread de usuario, mientras que los Objetos Activos sólo usan unos pocos bytes. Otra ventaja es que los Objetos Activos consumen mucho menos recursos durante la conmutación y el proceso mismo.

La diferencia de tiempo puede llegar a tener un factor de 10 en favor de los Objetos Activos. Es importante tomar esto en cuenta, porque influye directamente en la efectividad de la aplicación y del sistema Symbian también.

CActiveScheduler provee 2 funciones virtuales que pueden ser usadas si se desea agregar o personalizar la implementación de la clases CActive. Si RunL() es llamada por el Active Scheduler durante un error y éste no es manejado por la función RunError() entonces Error(Tint) es llamada. Por defecto, se le da a la función la responsabilidad de cerrar la aplicación de la manera más apropiada posible.

El Active Scheduler acarrea los eventos secuencialmente. Cuando la petición de ejecución de un nuevo evento se realiza mientras éste está corriendo, por defecto, el active scheduler espera

finalizar el primer evento antes de ejecutar el segundo.

De todas maneras, es posible y reservado para programadores más avanzados modificar el método de horarios de los objetos activos.

3.6 La API de Streaming y el MMF

La API de Audio Streaming es la interfaz para realizar stream de muestras de audio desde y hacia el controlador de audio de bajo nivel, que es parte del Multimedia framework (MMF).

El audio transmitido es enviado y recibido incrementalmente. Esto significa que:

- Los clips de sonido enviados al controlador de bajo nivel (audio play) pueden ser enviados inmediatamente apenas son recibidos en vez de tener que esperar hasta que el clip completo sea recibido.
- El usuario de la API puede mantener los fragmentos de datos en una cola antes de enviarlos al servidor. Si el usuario intenta transferir los datos más rápido de lo que el servidor puede recibirlos, el exceso de fragmentos de datos son mantenidos en otra cola en el lado del cliente (invisible para el usuario), cuyos elementos son referencias a los buffers enviados. El servidor notifica al cliente usando una notificación de evento (callback) cada vez que ha recibido un fragmento. Esto indica al cliente que el fragmento puede ser borrado.
- Los clips de sonido que están siendo capturados por el controlador de bajo nivel (audio record) pueden ser leídos incrementalmente sin tener que esperar a que la captura de audio se complete.

El controlador de bajo nivel de audio, mantiene los buffers recibidos en un lugar donde puede almacenar el audio que está siendo capturado. El cliente usa una función de lectura para leer los datos recibidos en los descriptores de destino.

El cliente también es notificado (en ambos, audio play y record) cuando el stream es abierto y está disponible para su uso (la abertura se hace asíncronamente), y cuando el stream es cerrado.

Esta API sólo puede ser usada para stream de datos de audio, con los datos guardados o provistos desde un descriptor. La API no soporta mezclas. Se debe usar un mecanismo extra para controlar el acceso al dispositivo de sonido por más de un cliente.

3.6.1 Uso de CMdaAudioInputStream

Es usado para accesar a los decodificadores de `DevSound` directamente. Esta API actúa como una API transparente, es decir, todos los decodificadores disponibles a través de `DevSound` están disponibles a través de `CmdaAudioOutputStream`.

El uso de la interfaz involucra la apertura, el ajuste de las propiedades de audio, la escritura y el cerrado del stream. Estas acciones son implementadas por las clases `CmdaAudioOutputStream` y `MMdaAudioOutputStreamCallback`.

Cada etapa se describe a continuación:

- Se crea un nuevo objeto de streaming de audio, usando `NewL()` y opcionalmente se define la prioridad que tendrá el streaming con respecto a otros clientes que intenten usar el hardware de audio.
- Para abrir el stream se usa `Open()`. Una vez que el *stream* ha sido abierto con éxito, `MaoscOpenComplete()` es llamado para indicar que el stream está listo para ser usado.
- Para ajustar las propiedades de audio en el equipo, por ejemplo, el número de canales de audio o el sampling rate se usa `SetAudioPropertiesL()`. Los valores deben ser especificados como valores de enumeración, por ejemplo, `TMdaAudioDataSettings::ESampleRate8000Hz` en vez de 8000. No es posible reajustar estos valores una vez que el *stream* está siendo reproducido. Las funciones de *volume* y *balance* permiten ajustar estas propiedades y pueden ser usadas

mientras el *stream* está abierto, tomando efecto inmediatamente.

- `WriteL()` sirve para especificar qué buffer se enviará a las capas más bajas del MMF. Una vez que el buffer ha sido copiado exitósamente, un puntero a su ubicación es retornado a `MaoscBufferCopied()`. Al entrar en este evento el buffer puede ser borrado, pues ya no será usado nuevamente. También en este evento existe la posibilidad de efectuar nuevas llamadas a `WriteL()`, pues en todo momento el MMF mantiene su propia lista de buffer para reproducir.
- Para detener la reproducción se usa `Stop()`. Una llamada al evento `MaoscPlayComplete()` indicará el cerrado correcto del stream.

Nota: `MaoscPlayComplete()` puede también ser llamado, si es que no hay más datos de audio para reproducir. En estas circunstancias el stream de audio es cerrado automáticamente y la variable `aError` del evento vale `KerrUnderFlow`.

El uso de esta API enfrenta un trade-off entre el uso de pequeños o grandes buffers para almacenar los datos de audio. El uso de buffers pequeños, aumenta la probabilidad de un `underrun` (en donde el dispositivo finaliza la reproducción antes de que el siguiente buffer haya sido recibido), pero disminuye el retardo inicial del audio al ser reproducido.

3.6.2 Uso de CMdaAudioOutputStream

Análogamente la API `CMdaAudioInputStream`, también actúa transparentemente, es decir, permitiendo el acceso a los codificadores de `DevSound`.

El uso de esta interfaz considera la apertura, el ajuste de las propiedades de audio, la lectura y el cerrado del stream. Estas funciones son análogamente implementadas por las clases `CMdaAudioInputStream` y `MMdaAudioInputStreamCallback`.

Cada etapa de la uso de esta interfaz es descrito a continuación:

- Se crea un nuevo objeto de audio streaming usando `NewL()` y opcionalmente se define la prioridad que tendrá el streaming con respecto a otros clientes que intenten usar el

hardware de audio.

- Para abrir el stream se usa `Open()`. Una vez que el stream ha sido abierto con éxito, `MaiscOpenComplete()` es llamado para indicar que el stream está listo para ser usado.
- Para ajustar las propiedades de audio en el equipo, por ejemplo, el número de canales de audio o el sampling rate se usa `SetAudioPropertiesL()`. Los valores deben ser especificados como valores de enumeración, por ejemplo, `TMdaAudioDataSettings::ESampleRate8000Hz` en vez de 8000. No es posible reajustar estos valores una vez que el stream está siendo reproducido. Las funciones de *ganancia* y *balance* permiten ajustar estas propiedades y pueden ser usadas mientras el stream está abierto, tomando efecto inmediatamente.
- `ReadL()` sirve para especificar qué buffer se usará para grabar los datos de audio desde las capas más bajas del MMF. Una vez que el buffer ha sido escrito exitósamente, un puntero a su ubicación es retornado a `MaiscBufferCopied()`. MMF sólo comienza a grabar datos de audio no despúes de `Open()` sino luego de la primera llamada a `ReadL()`.
- Para detener la reproducción se usa `Stop()`. Luego de esto, dos eventos son llamados, el primero es `MaiscBufferCopied()` apuntando al buffer que contiene los últimos datos de audio grabados (`aError` toma el valor `KErrAbort`). El segundo es `MaiscRecordComplete()` indicando el cerrado correcto del stream.

3.7 El emulador y el Nokia N75

El IDE sugerido por Nokia para desarrollar aplicaciones ha sido implementado a partir de Eclipse y se llama Carbide. Para este trabajo se utilizó Carbide C++ que es el específico para trabajar en Symbian C++.

Cada kit de desarrollo o SDK trae consigo un emulador del dispositivo específico para la familia de terminales para los cuales se está desarrollando la aplicación. Las ventajas del emulador, radican en la posibilidad de hacer debug (corregir errores) sobre las variables y

funciones que están corriendo en nuestra aplicación de manera rápida y sencilla. Lamentablemente, la posibilidad de hacer esto con el prototipo final no es posible, porque el emulador no tiene las funcionalidades multimedia desarrolladas en un 100%, como por ejemplo, la funcionalidad de grabar en AMR (Adaptive Multi-Rate).

Dadas las limitaciones del emulador, se optó por desarrollar toda el sistema de simulación sobre grabaciones en PCM (Pulse-Code Modulation), corregirle los errores y luego adaptarlo en un dispositivo móvil para usar AMR. Las diferencias principales entre estos dos tipos de formatos se pueden apreciar en su estructura y tamaño; mientras cada muestra de audio en PCM pesa 4096 bytes, almacena 0.256 segundos de audio y no tiene encabezado, AMR pesa 14 bytes (ver los distintos tipos en la Tabla 4), almacena 0.02 segundos de audio y tiene un encabezado de tamaño un byte (el primero de los 14) que está explicado en la siguiente sección 4.1. En la figura 11 se puede apreciar una foto del emulador.

El principal dispositivo físico de prueba en este proyecto fue el teléfono celular de marca Nokia modelo N75, en adelante llamado Nokia N75. Es un teléfono 3G con formato plegable que se encuadra dentro de la gama musical de Nokia. Viene equipado con un reproductor capaz de sincronizar con Windows Media Player y reproducir MP3, AAC, eAAc+ y WMA. Incluye además Radio FM y altavoces estéreo con sonido 3D. Tiene incluidos por hardware los codecs AMR, FR, EFR y HR.

También posee un slot para tarjetas de expansión de memoria microSD, 60 MB de memoria interna compartida, cámara digital de 2 megapixeles, cliente de correo electrónico y bluetooth 2.0.

Para comprobar los resultados se utilizó también el modelo Nokia N73, que tiene el mismo sistema operativo instalado y tiene las carecterísticas multimedias apropiadas para correr la aplicación desarrollada. En la figura 12 una foto del Nokia N75.

Figura 11: Emulador S60 3rd Edition FP1 SDK for Symbian OS

Figura 12: Nokia N75

CAPÍTULO 4: IMPLEMENTACIÓN

Para poder tomar las decisiones pertinentes en la implementación, se estudió el comportamiento de la representación numérica de muestras de audio resultantes en grabaciones de conversaciones cotidianas en el idioma español. Se realizaron varias grabaciones y fueron graficadas en búsqueda de patrones. En la figura 13 se puede ver un ejemplo de esto.

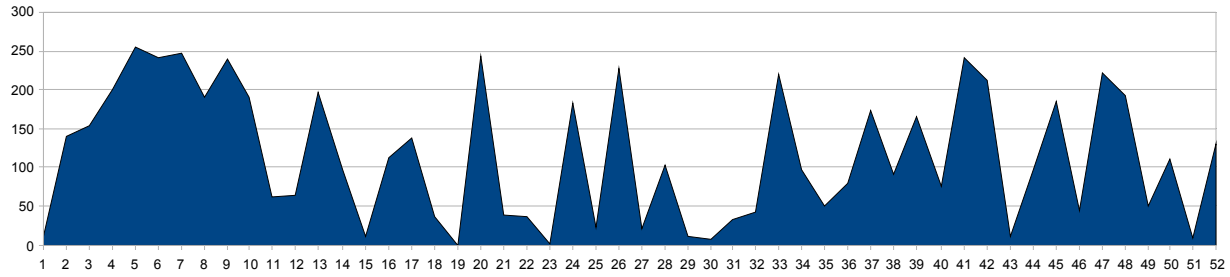

Figura 13: Representación numérica de 3 frames de una conversación en AMR

Como se puede ver, independiente de que los valores pertenezcan al rango [0,255] no se aprecian patrones que sugieran algún tipo de reconstrucción obvia, por esto es que se probaron varios tipos de reconstrucciones, para ver cuál era el que mejor desempeño tenía.

Para probar el desempeño de las restauraciones y obedeciendo a la necesidad de tener un ambiente controlado de pérdidas en tiempo real, es que se optó por recrear el sistema normal de streaming, el que consta de una secuencia lógica (en función del tiempo) de: grabación, codificación y empaquetamiento, envío, posibles pérdidas, recepción, decodificación, reconstrucción (en caso de ser necesario) y reproducción. Para simular este proceso se comenzó desde un proyecto de ejemplo de Nokia que permite tener acceso al micrófono y al auricular de un dispositivo, al cual se le modificaron alrededor de 5 archivos para poder interactuar con *AudioStreamCodec.cpp* que es el principal motor de la aplicación y que será detallado a continuación.

Antes de entrar en detalle al ciclo de codificación y decodificación, es bueno saber que se trabajó con el tipo de datos `TBuf8`, el cual desciende de `TDes8` y es un descriptor que provee un buffer de tamaño fijo – cuyos datos se almacenan en el *stack* – que permite contener, accesar y manipular información del tipo `TUint8` (entero de 8-bit sin signo independiente de la implementación). Existe también un tipo de datos – cuyos datos se almacenan en el *heap* – llamado `RBuf8`, el cual tiene una serie de beneficios considerando lo escaso de los recursos en los teléfonos móviles y porque puede ser creado dinámicamente. Lamentablemente `RBuf8` puede ser sobreescrito una vez que es usado y para efectos del estudio era necesario comparar las grabaciones originales y las reconstrucciones hechas, por lo que se necesitó de un tipo de datos persistente durante la ejecución de la aplicación. Ésta es la razón de porqué se utilizó `TBuf8` y no `RBuf8`.

4.1 Grabación

Para la grabación se utilizó la API CMdaAudioInputStream. Como ya se mencionó esta API se comporta como un objeto activo, el cual corre hasta encontrarse con un evento que determine su detención. Entonces este ciclo consiste básicamente en grabar un múltiplo de 16 muestras de audio, llamadas individualmente ***Frames,*** comprimidas por hardware con el codec AMR. Cada lectura del micrófono con el codec AMR retorna una lectura de 20 ms (0.02 segundos) de audio, almacenados en 14bytes y que corresponden a $8000\text{Hz} * 16\text{bits} * 0.02\text{seg} = 2560\text{bits} = 320\text{bytes}$ en PCM, es decir, sin compresión. Los 14 bytes vienen del tipo del nivel de codificación usado según se muestra en la tabla 4:

CMR	Mode	Frame size (bytes)
0	AMR 4,75	13
1	AMR 5,15	14
2	AMR 5,9	16
3	AMR 6,9	18
4	AMR 7,4	20
5	AMR 7,95	21
6	AMR 10,2	27
7	AMR 12,2	32

Tabla 4: Tipos de codificación en AMR

El dispositivo por defecto comprime con el modo AMR 5,15, lo que determina que cada frame de AMR tenga una estructura definida también, que debe ser replicada al momento de ser restaurada. El primer byte contiene un header y los restantes, en este caso 13 bytes, contienen la información de audio. Los primeros 4 bits del header contienen el valor CMR (Codec Mode Request), válido entre 0 y 7, los últimos 4 bits, son reservados y no son usados. [35]

Recordar que una lectura al micrófono en el emulador y/o sin compresión, es decir en PCM, retorna un buffer de 4096 bytes, lo que representa a 0.256 segundos de audio.

4.2 Codificación

Se escogió grabar un múltiplo de 16 muestras, en total 256, para poder estudiar la reconstrucción explicada en la sección 2.3 llamada *Aleatorización con interpolación.* Para realizar las pruebas, la función de codificación permite empaquetar los frames de dos maneras, secuencial o intercalada y los 16 frames entonces, son empaquetadas por esta función en 4 ***buffers*** que contienen 4 ***frames*** de audio cada uno.

`void CAudioStreamCodec::Encode(TUint aMode)`: Recibe de parámetro el modo de codificación a utilizar. Los modos son:

- `LINEAL`: Se crean buffers en el orden que se capturan. Esto es linealmente en el tiempo: buffer0 (0,1,2,3), buffer1 (4,5,6,7), buffer2 (8,9,10,11) y buffer3 (12,13,14,15). Este formato tiene en teoría[iii] la ventaja de que se pueden empezar a enviar datos desde la captura del frame 4, es decir, luego de que se completa la codificación del primer buffer, lo cual disminuye la latencia en la recepción, la que vendría dada por el tiempo que se toma en grabar 4 muestras de audio más el tiempo que demora el paquete en llegar a destino. El lado negativo es que si se pierde algún buffer en el envío, se pierde mucha cantidad de información que por ende, es más difícil de reconstruir.
- `RANDOM`: En este modo se crean buffers, según la técnica de *Aleatorización con interpolación*. Gráficamente se crean: buffer0 (0,4,8,12), buffer1 (1,5,9,13), buffer2 (2,6,10,14) y buffer3 (3,7,11,15). Al contrario del modo *lineal* de codificación este modo

iii El sistema de decodificación y reconstrucción explicado más adelante hacen que sea imposible aprovechar las ventajas de enviar los paquetes secuencialmente y aminorar la latencia, pues se espera la llegada de un paquete completo antes de iniciar su reconstrucción.

tiene en teoría[iv] la desventaja de que hay que esperar no menos de 12 frames de audio para poder enviar un paquete. Esto produce una latencia no inferior a $12\text{frames} * 0.02\text{seg} = 0.24\text{seg} = 240\, \textit{milisegundos}$ y de acuerdo a lo expuesto en los capítulos anteriores, el oído humano puede detectar en promedio, latencias superiores a 250ms. Por esto, no se consideraron más frames para conformar un paquete, dado que además de la compresión se debe considerar un tiempo correspondiente al envío de cada paquete.

4.2.1 El header

Dado que cada paquete contiene 16 frames de audio encapsulados en 4 buffers, inicialmente se pensó utilizar un entero (`TUint8`) como header puesto que éste está formado por 4 bytes (32 bits) y con eso se podía tener la información considerada necesaria para poder realizar la reconstrucción de manera simple y efectiva. Para esto cada frame debía contener distribuida la siguiente información:

- Sequence Number: indica el número de secuencia de cada frame. Cada paquete está conformado por 16 frames enumerados del 0 al 15, así que se asignó el primer byte (8 bits) para este fin, lo cual en teoría permite tener hasta 64 frames en cada paquete, para poder experimentar. En la práctica más de 16 frames por paquete pueden producir un delay intolerable en la comunicación.
- Packet Number: el envío de paquetes también está secuenciado, para mantener el orden de la muestra de audio. Se asignaron los 3 bytes restantes, los que en sus 24 bits permiten enviar $2^{24} = 16777216$ paquetes secuencialmente. Luego de esa cantidad de paquetes, el número se reinicia como contador en 0.

La estructura de cada header lucía como se puede apreciar en la figura 14:

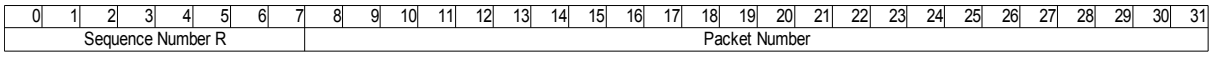

Figura 14: Idea original de encabezado

Esta codificación es muy sencilla de programar en tiempo real, pues es cuestión de escribir el header justo antes de realizar la captura del micrófono y así, secuencialmente, ir anexando uno

iv Idem al tipo LINEAL explicado en el punto iii.

a uno los 4 frames de cada buffer, para finalizar el ciclo enviándo el buffer a su destino. Sin embargo, enfrenta 2 aspectos negativos, que se buscaron solucionar:

1. es completamente innecesario repetir el número de paquete, *Packet Number*, al inicio de cada frame.
2. es confuso analizar un buffer que tiene mezclados encabezados y audio.

Para evitar repetir información y centralizar todos los datos en un solo encabezado, se refinó la versión anterior incluyendo ahora información sobre qué tipo de decodificación se quiere realizar, seguido del número de paquete para luego dar paso a los 4 frames que se incluyen en cada buffer.

Así, el encabezado terminó finalmente como se aprecia en la figura 15:

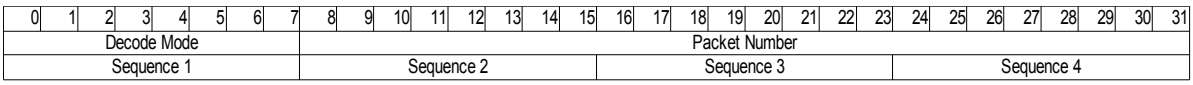

Figura 15: Encabezado de cada buffer

Donde:

- Decode Mode: es una bandera que puede indicar de qué manera fueron codificados los frames o con qué técnica debe hacerse la reconstrucción o ambas, por ejemplo, si debe hacerse reemplazando por ceros los frames perdidos, repitiendo el último frame recibido, interpolando linealmente, interpolando sinusoidalmente, etc. Este indicador da la posiblidad de implementar numerosas técnicas de restauración frente a las pérdidas sin realizar cambios al resto del sistema y eventualmente, escoger la mejor de éstas de manera en tiempo real según: el porcentaje de pérdida, la dinámica de la muestra de audio, si es voz o música, o según algún otro factor de decisión, por nombrar algunos ejemplos.
- Packet Number: el envío de paquetes es secuenciado, para mantener el orden de la muestra de audio, por lo cual esto no se modificó. Se mantienen los 3 bytes, es decir, 24 bits que permiten enviar $2^{24}=16777216$ paquetes secuencialmente. Si se alcanzara a enviar esta cantidad de paquetes, este número puede ser reiniciado en 0 nuevamente.
- Sequence Number: igual que en el modelo anterior, indica el número de secuencia del frame, con la diferencia que este encabezado incluye al comienzo de cada buffer la información sobre los 4 frames que trae en su interior, que están enumerados del 0 al 15. Sequence 1, corresponde al número de secuencia que le corresponde al primer frame de audio en el buffer y así correlativamente, Sequence 2 al segundo, Sequence 3 al tercero y

Sequence 4 al cuarto.

Como ejemplo de encabezado consideremos la muestra: (167773909,17107213) que si lo analizamos según las secciones de un header postuladas en la figura 15 su decodificación sería: (10, 1749, 1, 5, 9, 13) números que junto con establecer que se debe decodificar con el tipo 10 (reemplazos por ceros), indica también que recibiremos los frames 1, 5, 9 y 13 del paquete 1749.

Si bien el header es dependiente de la cantidad de frames que conforme un paquete (mientras más frames, más bytes se agregan) la relación de un byte para representar un frame de audio de 14 bytes es bastante apropiada para hacer streaming. En este caso un buffer tiene 4 frames, lo que equivale a 56 bytes, mientras que el header alcanza apenas los 8 bytes, menos del 15% del tamaño del buffer. Los métodos *Append* y las lecturas del tipo *buffer*[*indice*] permiten anexar y leer exactamente un byte, por lo que la codificación escogida es simple de programar y entender, pero es bueno mencionar, que existe la posibilidad de optimizar aún más este encabezado, si se codifican en 4 bits el método de decodificación y cada frame que viene en cada buffer (pues los frames van numerados del 0 a 15 y no se necesitan más de 4 bits para representarlos). Esto nos dejaría el header actual en 5 bytes, correspondientes a menos del 10% de cada paquete.

4.3 Simulación de pérdidas

Como ya se ha dicho en capítulos anteriores, los paquetes enviados por UDP (protocolo utilizado preferentemente para hacer streaming) pueden resultar perdidos dependiendo del tráfico de la red y/o de muchos otros factores. Así pues, para recrear esto, se implementó un método que simula la pérdida de algunos buffers, ya sea en ráfaga o aleatoriamente.

`void PacketLossSimulator(TUint aMode, TUint anAmount)`: Recibe de parámetro además del modo, una cantidad de buffers que se debe simular perder. Los modos son:

- `BURSTY`: Este método simula pérdidas en ráfaga, esto es, se escoge un paquete inicial al azar y se borran tantos buffers como correspondan de acuerdo a la cantidad recibida en `anAmount`. Si el buffer inicial más la cantidad de buffers calculadas a borrar exceden el

tamaño del buffer codificado completo, entonces cíclicamente se borran buffers del comienzo. En la figura 16, se puede apreciar el comportamiento de esta función.

Se envían 8 paquetes de 4 bufferes cada uno. Un buffer recibido se representará con una 'O' y una pérdida con una 'X'

Packet1	Packet2	Packet3	Packet4	Packet5	Packet6	Packet7	Packet8
O O O O	O O O O	O O O O	O O O O	O O O O	O O O O	O O O O	O O O O

Se escoge al azar un paquete, en este caso el 7 y se pierden `AnAmount` buffers en los próximos paquetes. En este caso `anAmount` = 9.

Packet1	Packet2	Packet3	Packet4	Packet5	Packet6	Packet7	Packet8
X X X O	O O O O	O O O O	O O O O	O O O O	O O O O	O O X X	X X X X

Figura 16: Ejemplo de Pérdida en ráfagas

- `RANDOM`: Simula pérdidas aleatorias para tantos buffers como correspondan de acuerdo al número indicado en el argumento `anAmount`. Se debe notar que en este caso los buffers eliminados pueden ser de cualquier paquete. En la figura 17 se puede apreciar el funcionamiento de este método.

Packet1	Packet2	Packet3	Packet4	Packet5	Packet6	Packet7	Packet8
O O O O	O O O O	O O O O	O O O O	O O O O	O O O O	O O O O	O O O O

Por cada una de las `anAmount` iteraciones de un ciclo, se escoge un paquete al azar y se eliminan los buffers correspondientes. En este ejemplo `anAmount` = 6.

Packet1	Packet2	Packet3	Packet4	Packet5	Packet6	Packet7	Packet8
O O O O	O X X O	O O O O	X O X O	O X O X	O O O O	O O O O	O O O O

Figura 17: Ejemplo de Pérdidas aleatorias

- `CUSTOM`: Permite eliminar cuántos buffers se deseen desde cualquier paquete.

Al cuestionarse porqué estas distintas simulaciones fueron implementadas, se descubre que aportan distinta información sobre la reconstrucción de la voz. Las pérdidas en ráfaga ayudan a comprender hasta cuántos buffers se pueden perder sin que se pierda el contexto de una

conversación y qué reconstrucción es mejor aplicar según el caso, mientras que las pérdidas aleatorias son más frecuentes al realizar streaming y por lo tanto aporta datos de cómo podrían resultar las reconstrucciones en una aplicación real. Las pérdidas controladas en el modo `CUSTOM` permiten tener posibilidades mixtas entre ráfagas y aleatorias.

4.4 Simulación de Recepción – Reconstrucciones

Una vez codificados los paquetes y enviados al lado receptor, deben ser recibidos primero antes de ser decodificados. Para simular esto, se utilizaron 2 métodos que forman en conjunto un ciclo: un recibidor de buffers y un decodificador.

`void CAudioStreamCodec::BufferReceiveSimulator(TUint aMode)`: se dedica a administrar la llegada de buffers, los que pueden llegar o no (en caso de pérdida) o incluso llegar en desorden. En la recepción de cada buffer, los frames recibidos son ordenados en un arreglo (llamado `iAuxPacket`) y marcados como recibidos en otro (llamado `iHasContent`). Para realizar esto, consta de un ciclo el cual, mientras recibe buffers del mismo paquete (según diga la componente `iPacketNumber`), se dedica a ordenarlos y para luego decodificarlos. Cuando se recibe un buffer del siguiente paquete o cuando se alcanza la cantidad de buffers necesaria (4 en este caso, y ocurre como caso de borde al final de la muestra) el paquete completo es pasado al decodificador. La tolerancia del sistema aguanta que buffers del mismo paquete lleguen en desorden, pero no de distintos paquetes, esto pues según ya se ha explicado la latencia para soportar desorden entre distintos paquetes puede ser intolerable al oido humano.

Decode es el proceso que realiza la decodificación restaurando los frames perdidos. Para eso recibe el arreglo `iAuxPacket` (16 frames) desde `BufferReceiveSimulator` y le aplica el proceso de restauración seleccionado en los casos negativos (`EFalse`) que indique el arreglo `iHasContent`, para luego encolar el paquete completo reconstruido a la lista de reproducción. En la figura 18 se puede ver una gráfica del ciclo Recepción-Decodificación.

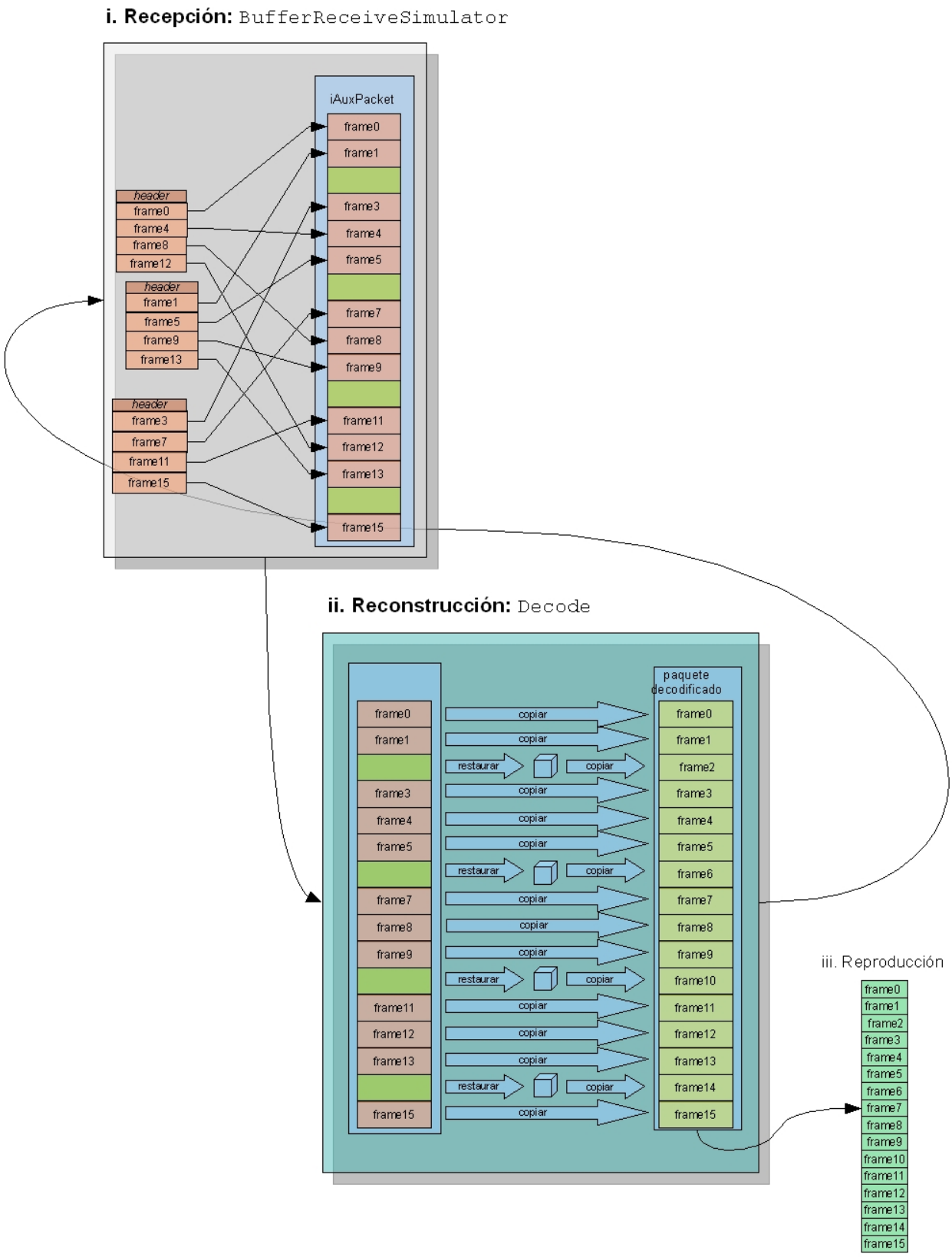

Figura 18: Ciclo Recepción-Reconstrucción

`void CAudioStreamCodec::Decode(TUint aMode)`: Recibe de parámetro el modo de decodificación a utilizar.

- `ZERO_REPLACE`: Reemplazo por ceros. En la detección de un paquete perdido, éste se reemplaza por ceros (valor 0 en PCM y 255 en AMR), lo que en audio se traduciría en un paquete sin sonido o en silencio.
- `REPEAT_LAST`: Reemplazo con el último recibido. Acá un frame de un buffer perdido es restaurado copiando el frame inmediatamente anterior recibido.
- `REPEAT_BEST`: Para realizar este tipo de reemplazo, se tomó la idea exhibida en la sección 2.3 *Repetición por Planilla*. Esta idea supone condiciones y requisitos que no necesariamente se dan en y entre teléfonos móviles[v] además de usar un índice en el que se basa su rapidez y eficiencia. Dadas las limitaciones de estos dispositivos ya expuestas, se tomó sólo la idea de buscar el mejor frame y se definió entonces, que para escoger los datos que reemplazarían al frame perdido se tomaría el último byte recibido y se buscaría un calce dentro de todos los bytes recibidos del paquete en reconstrucción, para luego copiar los frames siguientes al calce en el frame perdido. Cada byte puede contener información numérica en el rango [0,255] y en el mejor caso (que se pierda un solo buffer) se cuenta con $14\text{bytes} * 3\text{frames} = 42\text{bytes} - 3\text{bytes}\,de\,header = 39\text{bytes}$ de información, por lo que, no se puede esperar encontrar exáctamente el byte buscado, por esto definió una variable de holgura de valor ± 10, y que permite que se busquen calces entre el valor buscado y ± 10. En una variable se almacena la posición en que se encuentra el calce o la posición de menor diferencia con respecto al valor buscado. En caso de no haber calces, la posición de menor diferencia es usada. Así se puede lograr este proceso en un sólo recorrido secuencial de los frames recibidos. El proceso completo se explica en la figura 19:

v Los supuestos son que la señal es periódica, que se posee la memoria suficiente para almacenar un trozo de la señal y que con probabilidad cercana a 0 se perderán dos paquetes IP seguidos.

Frame0	12	198	60	199	255	240	247	183	239	127	62	30	230	6
Frame1														
Frame2	12	255	24	199	255	250	255	186	191	63	110	13	25	204
Frame3	12	108	60	199	255	241	255	183	207	191	207	188	59	40

Se ha perdido el frame 1, entonces se busca el último byte recibido, en este caso corresponde al Frame0 y tiene valor 6. Se recorre el paquete buscando valores entre 6±10, es decir, entre 0 (límite inferior) y 16 encontrando un calce en el Frame2 con valor 13.

Frame0	12	198	60	199	255	240	247	183	239	127	62	30	230	6
Frame1														
Frame2	12	255	24	199	255	250	255	186	191	63	110	13	25	204
Frame3	12	108	60	199	255	241	255	183	207	191	207	188	59	40

Se escribe el header y luego se copian los 13 bytes posteriories al calce y finaliza la reconstrucción.

Frame0	12	198	60	199	255	240	247	183	239	127	62	30	230	6
Frame1	12	25	204	108	60	199	255	241	255	183	207	191	207	188
Frame2	12	255	24	199	255	250	255	186	191	63	110	13	25	204
Frame3	12	108	60	199	255	241	255	183	207	191	207	188	59	40

Figura 19: Reconstrucción Repeat Best

- `MEAN_INTERPOL`: En este caso, para restaurar un frame de un buffer perdido, se realizan 3 cálculos:
  1. `last_media`: calcula el promedio numérico de los últimos 3 bytes del frame anterior (esto es byte[11], byte[12] y byte[13]).
  2. `next_media`: calcula el promedio numérico de los primeros 3 bytes de audio del frame siguiente (es decir byte[1], byte[2] y byte[3], pues byte[0] contiene el encabezado).
  3. `media`: es el resultado de promediar last_media y next_media.

  Finalmente el frame perdido se rellena con el valor que se obtiene en `media`.
- `RAND_INTERPOL`: El frame perdido es llenado aleatoriamente con distintos valores.

Luego de varias pruebas de esfuerzo se tuvo que bajar la complejidad propuesta para los métodos `REPEAT_BEST` y `MEAN_INTERPOL`. Esto porque el dispositivo utilizado no lograba llevar a cabo reconstrucciones con pérdidas sobre el 20% y/o en condiciones de ráfaga en los que se perdían varios buffers, pues para realizar estas restauraciones se debe encontrar el próximo frame, y bajo un sistema de altas pérdidas éste puede no existir y el algoritmo para encontrar algún siguiente frame con información hacía que el dispositivo dejara de funcionar. Las dos

modificaciones realizadas se detallan a continuación:

- En `REPEAT_BEST` luego de encontrar un calce, se copian – a partir del byte siguiente al calce – 13 bytes cíclicamente del mismo frame. Por ejemplo, si el calce está en la posición 11, se parte copiando desde la 12, para luego seguir con la 13 y de ahí se copia desde el byte 1 en adelante (recordar que el byte 0, contiene el header de codificación de AMR). En la figura 20 un ejemplo.

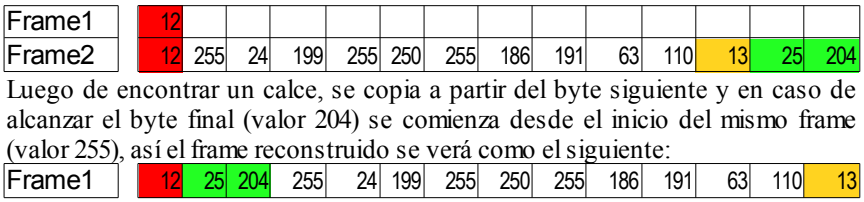

Figura 20: Modificación a REPEAT_BEST

- En `MEAN_INTERPOL` si el frame anterior o posterior al frame perdido no contiene información, entonces se asigna como previo y/o próximo frame, una muestra vacía de audio. Así se pueden realizar los 3 cálculos sin problemas.

Se debe mencionar, por último, que decode está implementado en forma de módulos, por lo que cualquier nuevo tipo de reestructuración puede ser probado con poco esfuerzo extra.

4.5 Reproducción

Para la reproducción se utilizó la API CMdaAudioOutputStream. Análogamente a CMdaAudioOutputStream, esta API también se comporta como un objeto activo, el cual corre hasta encontrarse con un evento que determine su detención. Entonces la reproducción se lleva a cabo pasándole al sistema el paquete reconstruido en ciclo anterior de Recepción – Reconstrucción.

CAPÍTULO 5: RESULTADOS Y CONCLUSIONES

5.1 Software Obtenido

Los resultados de este proyecto se pueden resumir en 2 grandes logros: una aplicación de simulación de un ciclo de VoIP completamente funcional para dispositivos móviles, con métodos para capturar voz, codificar, simular el envío y las pérdidas, reconstruir y finalmente reproducir los buffers de audio recibidos. También se logró un prototipo de codec de audio que soporta pérdidas de paquetes, entendiéndose paquete por buffers enviados y no por paquetes según la notación *frame-buffer-paquete* utilizada en el capítulo anterior.

La máquina de simulación de pérdidas fue escencial para comprender el comportamiento de una muestra de audio, pues en una conversación la manera en que se representan los datos de voz varía según la compresión que se utilice. También fue posible estudiar las limitaciones del celular, pues en la implementación hubo que relajar algunas variables y ciclos pues el dispositivo no era capaz de procesarlos.

Dado que los resultados de una restauración pueden ser escuchado una y otra vez junto a la muestra original de audio, es que se sometieron estas muestras a una evaluación según el modelo MOS descrito en la sección 1.4 de este documento, adaptado de manera en que no se considera el valor aportado por el tiempo de envío del paquete dentro de la latencia, pues las evaluaciones se realizaron bajo un ambiente de simulación. La evaluación comprendió a un grupo de 20 personas escogidas al azar, sin previos estudios en audio, telefonía o redes, que escucharon 14 ejemplos de citas en español sometidas a distintos tipos de pérdidas y reconstruidas por los 5 tipos de restauraciones expuestos en el capítulo 4. Las citas pueden ser leidas en el *Anexo A*.

Los tipos de pérdidas simulados son:

- Cíclicas: se pierde un buffer o un grupo de buffers repetitivamente cada cierto tiempo constante.

- Aleatorias: este tipo de pérdidas no tiene un comportamiento definido, las pruebas se realizaron eliminando buffers al azar.
- Ráfagas: cuando se produce una pérdida, se pierden varios buffers seguidos.

En la tabla 5 se puede apreciar el tipo de pérdidas al que fueron sometidas las distintas muestras, con ejemplos que van desde el 10% hasta el 50% de pérdidas. La cantidad de buffers perdidos (sobre un total de 40), así como el porcentaje de pérdida para cada prueba se pueden apreciar en la tabla 6.

		MATRIZ DE PÉRDIDAS																																										
	N°	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
CICLICAS	1			X								X								X							X								X							
	2			X				X				X				X				X				X				X				X				X				X		
	3			X	X			X	X			X	X			X	X			X	X			X	X			X	X			X	X			X	X					
	4			X	X			X	X	X		X	X	X						X	X	X		X	X	X						X	X	X		X	X	X				
	5			X		X		X		X		X		X		X		X		X		X		X		X		X		X		X		X		X		X				
	6			X			X			X			X			X			X			X			X			X			X			X			X			X			
ALEATORIAS	7			X									X												X									X								
	8					X				X					X											X				X						X							
	9			X				X							X		X					X						X		X						X							
	10							X	X		X				X	X		X										X				X	X				X					
	11		X			X			X			X			X								X		X				X	X				X				X	X			
RAFAGAS	12													X	X	X	X	X	X	X	X																						
	13					X	X					X	X	X	X							X	X	X										X	X	X							
	14			X	X	X							X	X	X	X	X							X	X	X					X	X	X	X	X							

Tabla 5: Matriz de pérdidas

N°	Buffers Perdidos	% Pérdida
1	5	12.5
2	10	25
3	18	45
4	20	50
5	18	45
6	13	32.5
7	4	10
8	6	15
9	8	20
10	10	25
11	12	30
12	8	20
13	12	30
14	16	40

Tabla 6: Cantidad de buffers perdidos y porcentaje de pérdida.

El resultado de la evaluación MOS realizada se puede encontrar en la tabla 7 y en promedio la reconstrucción REPEAT_LAST es la que logra el mejor desempeño.

N°	1	2	3	4	5	6	7	8	9	10	11	12	13	14	MOS
ZERO	4.01	2.84	1.77	1.56	1.43	2.32	3.73	3.79	2.66	2.71	2.3	3.67	2.06	1.83	2.62
LAST	4.28	4.18	1.89	2.13	2.99	3.41	4.56	4.11	3.72	3.81	3.59	3.54	2.17	1.98	3.31
BEST	3.32	2.63	1.43	1.45	1.68	2.28	3.98	3.8	2.82	3.21	2.97	3.52	1.66	1.54	2.59
MEAN	3.01	2.39	1.27	1.22	1.21	1.72	3.63	3.73	2.33	2.48	2.31	3.54	1.49	1.38	2.26
RAND	2.71	2.14	1.19	1.11	1.14	1.72	3.47	3.58	2.23	2.46	2.12	3.53	1.51	1.35	2.16

Tabla 7: MOS detallado para cada ejemplo y reconstrucción

Importante es notar que, de este estudio se desprende que el prototipo de reconstrucción es evaluado positivamente y con una calidad aceptable si las pérdidas no superan el 30%, pues al eliminar estas pruebas de la medición alcanza un valor MOS de 4.03, lo cual se ve reflejado en la tabla 8. Aún más, el prototipo fue diseñado para pérdidas y desórdenes entre buffers que pertenecen al mismo paquete (ver Trabajo Futuro). La prueba 12 comprueba esto y es, efectivamente la que menor MOS aporta al codec, al contemplar la pérdida de un paquete completo (buffers del 12 al 15). Sin considerar esa prueba se alcanza un MOS de 4.11.

N°	1	2	7	8	9	10	12	MOS
ZERO	4.01	2.84	3.73	3.79	2.66	2.71	3.67	3.35
LAST	4.28	4.18	4.56	4.11	3.72	3.81	3.54	4.03
BEST	3.32	2.63	3.98	3.8	2.82	3.21	3.52	3.33
MEAN	3.01	2.39	3.63	3.73	2.33	2.48	3.54	3.02
RAND	2.71	2.14	3.47	3.58	2.23	2.46	3.53	2.87

Tabla 8: MOS para porcentajes de pérdidas menores al 30%

Finalmente, es interesante ver como los algoritmos más básicos de reconstrucciones obtuvieron un mejor resultado frente a los más complejos.

Así, el prototipo de códec fue logrado y se ha comprobado su funcionalidad en los terminales N73 y N75 de la serie 60 de Nokia.

5.2 Extensiones posibles

- Debido a las complejidades en el aprendizaje y desarrollo, que involucraba este sistema, junto con el poco tiempo de duración del proyecto no se realizaron pruebas en otros dispositivos para evaluar qué comportamiento y rendimiento tiene el prototipo. Con esto se podría pensar en extender las pruebas a otras marcas, otros modelos más limitados e incluso a otros lenguajes de programación.
- De acuerdo a los resultados obtenidos y dado que el primer byte del header puede indicar el tipo de reconstrucción que se utilizará, se podría extender este modelo para automáticamente escoger el mejor tipo de restauración y dejar que se autoajuste según las distintas variables: menor esfuerzo de cpu, mejor calidad, silencios, música, etc.

5.3 Trabajo futuro

- En cuanto a las estructuras de datos:
  - si se pensara en tener una aplicación real funcionando con este prototipo, la escasez de memoria que tienen de los dispositivos, sumado a la volatilidad de los paquetes indicaría que el tipo de datos a usar para almacenar los paquetes y buffers debe ser `RBuf8` pues son destruidos y liberados luego de su uso y no habría razón para mantener

en memoria por ejemplo un buffer ya escuchado, pero dado que RBuf8 almacena sus datos en el heap la aplicación podría caerse si es que al crease un RBuf8 nuevo para almacenar un nuevo buffer, no hubiera memoria disponible. Por esta razón, el tipo de datos TBuf8 usado para almacenar los buffers no debe ser cambiado, pues éste reserva espacio en el stack al iniciarse la aplicación asegurándonos un funcionamiento correcto durante su ejecución.

Aun así, dada la necesidad de escuchar varias veces los distintos tipos de reconstrucciones durante el desarrollo y evaluación de este proyecto, se utilizaron varios buffers del tipo TBuf8 para almacenar cada una de las distintas restauraciones. En una aplicación real, sólo se necesita un buffer para la reconstrucción, lo cual, haría que el programa funcionara con menos memoria de la que hoy utiliza, sin disminuir la capacidad de realizar varios y distintos tipos de restauraciones.

- Actualmente los buffers recibidos son decodificados y cada frame recibido se guarda en un arreglo de frames, necesitando así espacio en memoria para:
  - un arreglo de 16 frames
  - cada una de los frames que vienen en el arreglo
  - el buffer que almacena el paquete decodificado

  Se podría mejorar este método eliminando el arreglo y almacenando inmediatamente los frames en el descriptor (de tipo TBuf8) que será el paquete decodificado, realizando la reconstrucción sobre este mismo descriptor. Así se ahorraría nuevamente un espacio de la limitada memoria con la que cuentan estos dispositivos.
- En muchas situaciones se utilizó el comando *Append* para anexar datos a un buffer, comando que internamente además de agregar los datos correspondientes actualiza el tamaño del descriptor destino. Dado que se sabe de antemano el tamaño de los paquetes, es posible crear buffers del tamaño adecuado y simplemente copiar los datos con la función *Mem*::*Copy* evitándose así la instrucción extra e implícita que realiza Symbian en el manejo de sus descriptores, lo cual podría aminorar la latencia producida en un ambiente real.

- Sobre el desempeño:
  - La interpolación `MEAN_INTERPOL` puede ser mejorada con una interpolación polinómica, pues es más frecuente encontrar que el comportamiendo de la voz tenga algún patrón asociado, lo que determina que dentro de un frame hayan varios valores, en vez de ser reemplazada por un frame con un único valor en su interior. Esta reconstrucción se pensó y queda como trabajo futuro, pues su encabezo ya está incluido y listo para ser implementado como `POLY_INTERPOL`.
  - El principal reto en este proyecto era hacer las simulaciones y determinar qué reconstrucciones ofrecían mejor desempeño, pero para hacerse una idea indiscutible del desempeño de estas técnicas, la mejor prueba se daría en un ambiente real. Para esto lo que falta hacer es integrar el sistema codificador y decodificador con un prototipo ya diseñado de envío y recibo de paquetes, implementado durante la primavera del año 2007. La integración es bastante fácil, pues el prototipo de envío está basado en el mismo ejemplo de Nokia del cual se partió el proyecto codificador y éste último fue diseñado e implementado pensando en este futuro paso.

    La información básica necesaria sobre sockets y conectividad en Symbian puede ser encontrada en el *Anexo B*.
  - Hoy el prototipo soporta la pérdida y desórdenes entre buffers que tengan el mismo *packetNumber*, es decir, que pertenezcan al mismo paquete. Si se diera una pérdida en ráfaga en un ambiente real, se podrían perder 4 o más buffers consecutivos, perdiendo la totalidad de un paquete, que el prototipo hoy no se encarga de reconstruir. Los elementos para la reconstrucción están ya implementados, pues se cuenta con una variable que maneja el paquete actual que se está procesando, llamada `iPacketCurrentNumber` y una que alberga la información sobre qué paquete se está recibiendo, denominada `pnum` (proveniente de **p**acket **num**ber).

    La implementación de esta característica debe activarse justo antes de empezar el ciclo de reconstrucción y debe durar mientras `iPacketCurrentNumber` sea menor que `pnum`. Mientas dure esta condición debe encargarse de setear en falso cada

componente del arreglo que contiene la información sobre que frames tienen o no información, llamado iHasContent para luego dar paso al ciclo de restauración, lo cual generará un paquete con todos sus frames reconstruidos.

- El algoritmo de reconstrucción denominado REPEAT_BEST fue diseñado para ser rápido, dado que el buscar el mejor calce sobre un patrón es un proceso que debe considerar al menos mirar secuencialmente varios bytes y puede aumentar la latencia de una conversación a niveles intolerables, sin embargo, el prototipo implementado no considera la eventualidad de dos candidatos de calces, lo cual podría mejorar notablemente la reconstrucción. En la figura 21 se muestra un posible caso, en el cual si se consideraran, por ejemplo, los 3 últimos bytes del frame inmediatamente anterior al perdido, entonces la elección de qué información utilizar para el reemplazo cambiaría.

Frame0	12	198	60	199	255	240	247	183	239	127	62	30	230	6
Frame1	12													
Frame2	12	255	63	110	6	250	255	186	191	24	219	6	25	204
Frame3	12	108	60	199	255	241	255	183	207	191	207	188	59	40

Si se considera sólo el último byte, es imposible escoger entre dos posibles candidatos. En este ejemplo, los dos bytes con valor 6 en el Frame2 son posibles candidatos y podría ser decisivo mirar los valores anteriores para decidir.

Frame0	12	198	60	199	255	240	247	183	239	127	62	30	230	6
Frame1	12													
Frame2	12	255	63	110	6	250	255	186	191	24	219	6	25	204
Frame3	12	108	60	199	255	241	255	183	207	191	207	188	59	40

Figura 21: Ejemplo de reconstrucción REPEAT_BEST con 2 candidatos

Existen otros criterios de discriminación posibles que deben ser probados para evaluar el esfuerzo que requerirán en su procesamiento. No necesariamente se debe usar la comparación de los 3 últimos bytes, pues por ejemplo, puede considerarse el promedio entre el primer y útlimo byte del frame anterior o un vector que represente el estado de la curva de voz en ese momento para saber si es que va creciendo en su valor o decreciendo.

- Posibles integraciones:
    - Se puede integrar este prototipo con otro ya existente de video sobre IP para dispositivos móviles. [36]

BIBLIOGRAFÍA

1: Evolución e historia de la telefonía celular,
http://www.monografias.com/trabajos14/celularhist/celularhist.shtml

2: Celulares ¿quién no tiene uno todavía?,
http://www.familia.cl/colegios/celulares/celular.htm

3: Historia de VoIP,
http://voipex.blogspot.com/2006/04/historia-de-voip.html

4: Emerging Internet Voice Services: The Year Ahead, Christopher D. Libertelli,
http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=204

5: Voz sobre IP,
http://es.wikipedia.org/wiki/Voz_sobre_IP

6: Telefonia IP en Chile - VOIP Chile,
http://www.callip.org/cl/intro.php

7: What is VoIP,
http://www.voipgate.com/site/main/whatisvoip.php

8: Packet loss,
http://en.wikipedia.org/wiki/Packet_loss

9: Metodología para la medicion de latencia y pérdida en la red IP,
http://200.33.150.85:8080/idx_metodologia.php

10: Reconstruccion de Segmentos de Voz en Comunicaciones de voz por Paquetes,
http://veu.eupmt.es/publicacions/docs/ws.pdf

11: QoS (Quality of service),
http://www.voipforo.com/QoS/

12: VoIP en la Red del Operador,
http://www.aslan.es/boletin/boletin30/acterna.shtml

13: TECHtionary - MOS-Mean Opinion Score - VOIP,
http://www.techtionary.com/

14: El Estándar VoIP Redes y Servicios de Banda Ancha,
http://www.monografias.com/trabajos33/estandar-voip/estandar-voip.shtml

15: Session Initiation Protocol,
http://en.wikipedia.org/wiki/Session_Initiation_Protocol

16: Qué son codecs?,
http://mx.answers.yahoo.com/question/index?qid=20070127090116AAoxBjo&show=7
17: Telefonía Digital 3G,
http://es.wikipedia.org/wiki/3G
18: A Comparison of SIP and H.323 for Internet Telephony,
http://www.cs.columbia.edu/~hgs/papers/Schu9807_Comparison.pdf
19: IMS: IP Multimedia Subsystem,
http://en.wikipedia.org/wiki/IP_Multimedia_Subsystem
20: RTP: Real-time Transport Protocol,
http://es.wikipedia.org/wiki/Real-time_Transport_Protocol
21: User Datagram Protocol,
http://es.wikipedia.org/wiki/User_Datagram_Protocol
22: Adaptive multi-rate compression,
http://en.wikipedia.org/wiki/Adaptive_multi-rate_compression
23: Pulse-code modulation,
http://en.wikipedia.org/wiki/PCM
24: Preguntas Frecuentes VoIP - FAQ VoIP,
http://www.tsares.net/VoIP/FAQ_VoIP.htm
25: 2007-06-13 How many Skype users?,
http://skypenumerology.blogspot.com/
26: Codec,
http://en.wikipedia.org/wiki/Codec
27: Codec de audio,
http://es.wikipedia.org/wiki/C%C3%B3dec_de_audio
28: Reconstrucción de Segmentos de Voz en Comunicaciones de Voz por Paquetes,
http://veu.eupmt.es/publicacions/docs/ws.pdf
29: Audio Packet Loss over IP and Speech Recognition,
Mayorga, P.; Besacier, L.; Lamy, R.; Serignat, J.-F., 30 Nov.-3 Dec. 2003, Páginas 607 - 612.
30: Algoritmo de Reconstrucción de Señales Periódicas Transmitidas Vía IP,
http://www.dcc.uchile.cl/~jbustos/Pub/reconstruccion.pdf
31: Tutorial J2ME programación JavaMovil,
http://www.javamovil.info/J2ME/index.php
32: Symbian OS design faults,

http://www.codeproject.com/KB/mobile/Symbian_OS_design_faults.aspx

33: Exception Handling and Cleanup Mechanism in Symbian,
http://www.newlc.com/Exception-Handling-and-Cleanup.html

34: String and Descriptors,
http://www.newlc.com/String-and-Descriptors.html

35: Converting from AMR to PCM16,
http://www.newlc.com/topic-9211

36: Optimización de codec de video tolerante a fallas utilizando compresión espacial,
Felipe Lalanne, Memoria de Ingeniería Civil en Computación, Universidad de Chile, 2004.

37: Cual es la mejor letra de musica,
http://es.answers.yahoo.com/question/index?qid=20080205071506AAV3Np7

Anexo A: Citas usadas para evaluación MOS [37]

- “El paraíso lo prefiero por el clima; el infierno por la compañía.” Twain, Mark.
- “Por la calle del ya voy, se va a la casa del nunca.” Cervantes Saavedra, Miguel.
- “Las improvisaciones son mejores cuando se las prepara.” Shakespeare, William.
- “La gente te pide críticas, pero en realidad sólo quiere halagos.” Maugham, William Somerset.
- “En política, lo importante no es tener razón, sino que se la den a uno.” Adenaur, Konrad.
- “Saber cuando uno dispone de lo suficiente es ser rico.” Lao Tse.
- “De lo que no se puede resolver no se debe hablar.” Wittgenstein, Ludwig.
- “El destino es el que baraja las cartas, pero nosotros los que las jugamos.” Schopenhauer, Arthur.
- “Tres pueden guardar un secreto si dos de ellos están muertos.” Franklin, Benjamin.
- “La verdadera ciencia enseña, por encima de todo, a dudar y a ser ignorante.” Unamuno, Miguel.
- “Me lo contaron y lo olvidé; lo vi y lo entendí; lo hice y lo aprendí.” Confucio.
- “Quien de verdad sabe de qué habla, no encuentra razones para levantar la voz.” Da Vinci, Leonardo.
- “No hay camino para la paz, la paz es el camino.” Mahatma Gandhi.
- “Si necesitas una mano, recuerda que yo tengo dos.” San Agustín.

Anexo B: Conectividad en Symbian C++

La API Sockets Client tiene cinco interfaces cliente:

- `RSocketServ`: Establece y reserva recursos para la comunicación básica al socket server. Todas las otras interfaces necesitan que haya una sesión válidamente ya iniciada.
- `RConnection`: Es usada en el comienzo y en la administración de las conexiones activas. Aplicaciones que necesites manejar varias conexiones deben usar esta API. Es posible iniciar una conexión implícitamente, vía las APIs RSocket o RHostResolver, o explícitamente vía la API RConnection.
- `RSocket`: Punto de conexión para todas las comunicaciones basadas en sockets.
- `RHostResolver`: Hace las consultas de resolución de hosts.
- `RNetDatabase`: interfaz de acceso a bases de datos en la red.

La arquitectura de Sockets provee una interfaz cliente genérica y un servidor al cual se le pueden agregar varios módulos.

Servidor de Sockets

El socket server controla los accesos de los clientes a los servicios de sockets y provee información sobre los protocolos disponibles. La interfaz cliente del socket server está implementada por RsocketServ.

Cliente Socket

Un socket es un punto de conexión a un protocolo de comunicación. La interfaz cliente permite realizar las siguietes opciones con un socket: abrir, conectar, leer desde, escribir a, escuchar desde, aceptar de, fijar dirección y ajustar otras opciones.

La interfaz cliente es provista por Rsocket. Con `Rsocket`, la comunicación de datos usando el protocolo **TCP**, se realiza utilizando los métodos `Send(), Recv(), Read()` y/o `RecvOneOrMore()`. Para la comunicación de datos usando el protocolo **UDP**, se pueden

utilizar los métodos `SendTo()` para enviar datos a un host remoto y `RecvFrom()` para leer el primer datagrama encolado. En su retorno, la variable `anAddr` apunta al objeto `TInetAddr` que contiene la dirección remota fuente de los datos.

Nota 1: En UDP, si el socket no está asociado localamente a una dirección y un puerto, entonces la operación binding es realizada.

Nota 2: `RSocket::Listen` y `RSocket::Bind` son servicios síncronos, mientras que `RSocket::Accept` y `RSocket::Read` son operaciones asíncronas.