VNIVERSIDAD NACIONAL
AVTNOMA DE
MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA
DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA
DE LA COMPUTACIÓN

ANÁLISIS DE UN SISTEMA CONCURRENTE EN TIEMPO REAL PARA APLICACIONES BIOMÉDICAS

T E S I S
QUE PARA OBTENER EL GRADO DE:
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
FRANCISCO JAVIER CÁRDENAS FLORES

Dr. Fabían García Nocetti
Director
Dr. Héctor Benítez Pérez
Codirector

México D.F. Marzo 2008

...

Yo no supe dónde entraba
pero cuando allí me vi
sin saber dónde me estaba
grandes cosas entendí
no diré lo que sentí
que me quedé no sabiendo
toda ciencia trascendiendo.

...

Coplas hechas sobre un éxtasis
San Juan de la Cruz.

Dedicatorias

A la memoria de mi madre Nely,
por tu cariño incondicional...
A ti Papá que has cobijado con tus manos
mis esperanzas para el mañana...
Jime y Javi, esto es para que vean que las metas se alcanzan
no obstante el tiempo, siempre y cuando sean persistentes en
todo... *Los quiero...*
A ti Lauris por tu apoyo y tu amor incondicional...
Te quiero latocilla bonita. *Te amo.*

Agradecimeintos

*A*gradezco el apoyo brindado por el Proyecto de Cómputo de alto Rendimiento del "Macroproyecto Tecnologías para la Universidad de la Información y la Computación" de la Universidad Nacional Autónoma de México (UNAM).

Agradezco muy especialmente al Dr. Fabián García Nocetti por su apoyo en la dirección del presente trabajo de tesis. Gracias Fabián.

Agradezco las horas de discusiones de éste y otros temas al Dr. Héctor Benítez. Sin su invaluable ayuda este trabajo no culminaría. Muchas gracias Héctor.

Agradezco a los compañeros del DISCA-UNAM por todo el apoyo hacia el presente trabajo. Gracias compañeros.

Índice general

Índice de figuras

Índice de tablas

RESUMEN

En el presente trabajo se desarrollaron y analizaron dos aplicaciones para el despliegue del espectro de la señal *Doppler* ultrasónica buscando obtener tiempo real suave. Ambas aplicaciones se componen de tres tareas principales: captura, procesamiento y despliegue que están desacopladas mediante dos *Queues* concurrentes, además las tareas de captura y de procesamiento, son objetos activos (procesos asíncronos o hilos concurrentes). Se diferencian una de otra en la tarea de despliegue. En la primera aplicación se diseñó al objeto de despliegue como activo sin que se logre sincronizar el despliegue. En la segunda se logró la sincronización con el despliegue al insertar un generador en el ciclo principal para la atención de los eventos de la interfaz gráfica de usuario, lográndose así un comportamiento de tiempo real suave. Como se analizó a lo largo del trabajo, el reto fue obtener un muestreo en la lectura y escritura de los *Queues* concurrentes acorde a un despliegue síncrono por lo que la razón entre las lecturas y las escrituras del *Queue* concurrente es más del doble.

Capítulo 1

Introducción

El presente trabajo propone una metodología de programación concurrente para aplicaciones de procesamiento de señales en "tiempo real", en donde dichas aplicaciones estarán implementadas en una computadora del tipo personal con un Sistema Operativo (*SO*) dado, como *Linux* o *Mac Os X*, sin que los *SO*'s involucrados sean de *Tiempo Real*. Este trabajo se refiere al concepto de Tiempo Real Suave de Kopetz, [Kopetz, 1997], entendiéndose a este, como la aproximación del total de los tiempos de ejecución de las tareas involucradas hacia cierto tiempo límite. Las técnicas de programación concurrente están hechas con base en hilos, estructuras de datos avanzadas como los *Queues* concurrentes (*FIFOs* concurrentes) y eventos [Lutz, 2001].

Las aplicaciones de Procesamiento de Señales en Tiempo Real (PSTR) involucran: adquisición, procesamiento y despliegue, como se aprecia en la Figura 1.1. Al hablar de estos sistemas, típicamente se refiere a sistemas empotrados o embebidos con procesadores especializados para el procesamiento de señales DSP´s y convertidores A-D y D-A [Kuo et al., 2006]. Sin embargo, es importante destacar que no hay un *SO* involucrado.

En el presente trabajo, la captura, el procesamiento y el despliegue usan hilos y eventos que se ejecutan en forma concurrente en el *SO* en cuestión utilizando un intérprete. Esto da la garantía de portabilidad de las aplicaciones considerando el manejo de hilos y eventos.

El manejo de hilos lo proporciona directamente el sistema operativo, y se puede in-

Figura 1.1: **Esquema general para el procesamiento digital de imágenes en tiempo real.**

corporar a cualesquier lenguaje, sea lenguaje C, Fortran, etc. mediante bibliotecas. En el caso de los lenguajes interpretados el manejo de dichos hilos es mas elegante, pues es parte intrínseca a el, como el caso de *Python* [Lutz, 2001] y *Java* [Goetz et al., 2006].

Existen advertencias muy elocuentes en cuanto al uso de los hilos. En la década de los 1990´s, [Ousterhout, 1996] sugiere que se utilice el paradigma de eventos para hacer sobre todo aplicaciones de interfaces gráficas de usuario, y que solo se usen los hilos para el desarrollo de aplicaciones que sobre todo involucren cómputo científico. Recientemente Lee en [Lee, 2006b], y más detalladamente en [Lee and Zhao, 2007], demuestra la susceptibilidad de cometer errores en software dónde se utilizan hilos.

Sin embargo los hilos son la tendencia actual, por que la industria de la microelectrónica está ofreciendo varios microprocesadores en el mismo *Core*, debido a razones de disipación de calor. Se consume menos energía al tener dos (cuatro e incluso ocho) microprocesadores de 2 MHz en un mismo *Core*, que los microprocesadores de más de 4 MHz [Tanenbaum, 2006]. Por lo tanto para obtener aplicaciones que exploten los múltiples *Core* y obtener así los mejores desempeños, es necesario que las aplicaciones estén basadas en paradigmas concurrentes en base a hilos.

Partiendo de un análisis y diseño orientado a objetos, se propone el esquema de la Figura 1.2, en dónde se involucran los hilos y los eventos gráficos. La `Captura` y `Procesamiento` son objetos activos y se ejecutan asíncronamente, y tienen acceso a sus *Queues* concurrentes respectivos, los cuales se encargan de la sincronización de los hilos. El `Despliegue` que tiene que ver directamente con los *eventos* del sistema gráfico, es un punto medular del presente trabajo, debido al problema de como sincronizar éstos con los hilos asíncronos, la `Captura` y `Procesamiento`. Además con este esquema se da un desacoplamiento

entre las tareas de `Captura`, `Procesamiento` y `Despliegue` mediante los *Queues* concurrentes, dándose de esta manera independencia principalmente entre la `Captura` y el `Procesamiento`, de tal manera que si en un momento dado el despliegue, que es susceptible a entrar en un segundo plano debido a que en la computadora se atiendan cualesquier evento gráfico, no se interrumpan las acumulaciones de datos en el *Queue* a ser desplegados, tratándose de evitar de este modo la perdida de información. Aquí se supone idealmente que el tamaño de los *Queues* es infinita.

Otros grupos consideran al `Queue` como la vía de comunicación entre procesos, ver por ejemplo Data Flow Process Network [Lee and Parks, 1995].

Para manejar la interacción entre los hilos y los eventos, en el presente trabajo de tesis se estudian dos casos. En el primero se declara al objeto de despliegue, que intrínsecamente maneja los eventos del sistema gráfico, como un objeto activo. Esto es posible porque la API[1] utilizada para la creación de la interfaz gráfica de usuario (*WxWindows*) junto con el lenguaje interpretado utilizado (*Python*) hacen posible la creación del *"widget"*[2] para el despliegue, mediante la herencia múltiple de *clase Canvas*[3] de dicha API y la de los hilos. El segundo caso utiliza la creación de un *generador* o *iterador* citewikigenerator, el cual *Python* maneja [weightless threads, 2006]. Permitiendo hacer un planificador en la parte

[1]Del inglés Application Protramming Interface, Interfaz de programación de aplicaciones.

[2]"Widget" en el entorno de la interfaz gráfica de usuario, es un componente gráfico que permite la interacción con la aplicación. Como los botones, las ventanas, las barras de tareas, las cajas de texto, etc.

[3]En las interfaces gráficas de usuario se le denomina `Canvas` al "widget" que tiene las capacidades de graficar objetos como: lineas, círculos, imágenes, etc.

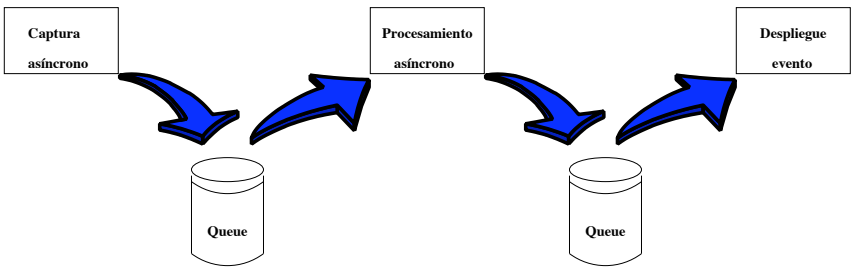


Figura 1.2: **Esquema para el procesamiento de señales en tiempo real utilizando *Queues* concurrentes para desacoplar las etapas de captura, procesamiento y despliegue.**

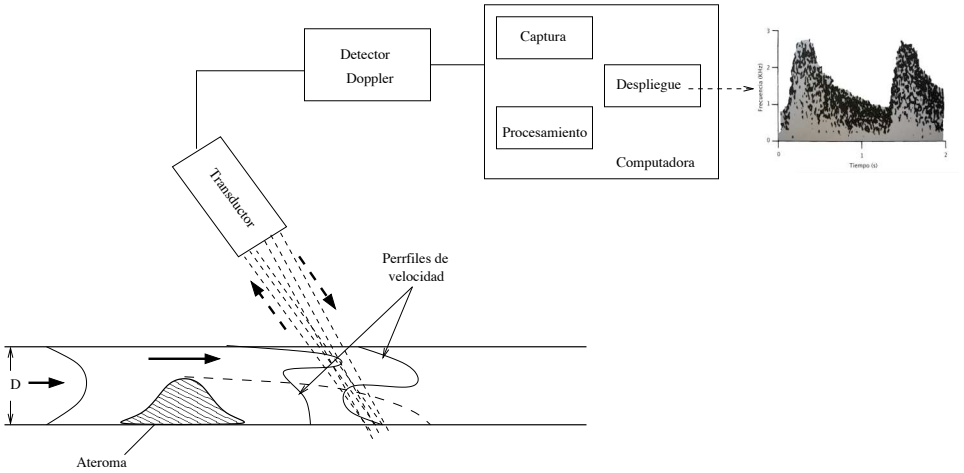


Figura 1.3: **Esquema del sistema de captura de señales *Doppler*.**

del manejo de eventos y así darle la sincronización adecuada a todo el procesamiento de señales en TRS.

La aplicación que se utiliza como caso de estudio, es un analizador de espectro para un sistema de *Doppler* ultrasónico utilizado en el diagnóstico del flujo sanguíneo. Hay dos tipos de sistemas para la detección de flujo sanguíneo, los de onda continua y los de onda pulsada, y en cada uno de estos tipos, existen desarrollos para la detección de flujo bidireccional [Fuentes et al., 2006, García et al., 2006]. Un esquema general para un sistema *Doppler* de onda continua se puede apreciar en la Figura 1.3.

El tipo de la señal *Doppler* capturada es *Estocástica Gaussiana Ciclo Estacionaria* en segmentos de 2-20 ms. como lo reportó Fish en su artículo [Fish, 1991]. Por lo tanto la señal se puede procesar digitalmente con una frecuencia de muestreo de 22 KHz. y una ventana de tamaño $N = 512$ (23 ms), ó con una ventana de $N = 256$ (11 ms) para obtener un resultado cuantitativamente aceptable para el diagnóstico angiológico. Se darán más detalles de estos valores en el capítulo 4.

1.1. Motivación

Desde que se comercializó la primer computadora del tipo personal, en la década de los 1980´s, el desarrollo de sistemas de diagnóstico médico en dichas computadoras, ha sido un tema de sumo interés para la comunidad de la ingeniería biomédica. En el campo del diagnóstico angiológico, como los equipos de ultrasonido *Doppler* para flujometría, dada la naturaleza audible de la señal *Doppler* ultrasónica, es atractivo hacer: adquisición, procesamiento, y despliegue del espectro de la señal, mediante una computadora portátil con equipo de reproducción y adquisición de audio, comercialmente asequibles.

La tecnología del hardware avanza mucho más rápido que el software por lo que es importante establecer métodos para desarrollar aplicaciones en Tiempo Real Suave transportables mediante lenguajes interpretados. Si un intérprete primero compila y proporciona una forma intermedia denomindada "bytecode"[Aho et al., 2007], dicha forma puede ser ejecutada por el intérprete en cualesquier sistema operativo y o arquitectura de hardware, si dicho intérprete esta debidamente instalado. Los mecanismos necesarios para el desarrollo de una aplicación concurrente para el despliegue del espectro de una señal *Doppler* ultrasónica son: manejo de hilos, estructuras de datos avanzadas como los *Queues* concurrentes, y el manejo adecuado de los *generadores* para la interacción directa con los eventos en el ciclo principal del sistema gráfico, de tal manera que se puedan controlar los eventos e hilos en la misma aplicación.

Por lo tanto la problemática es utilizar la concurrencia de manera tal que se logre tiempo real sin serializar los procesos.

1.2. Objetivo.

Estudio y desarrollo de un sistema concurrente mediante la sincronización de los procesos y eventos a través de *Queues* concurrentes para el análisis de señales en tiempo real.

1.3. Estructura del trabajo.

En el presente capítulo se ha dado una introducción general así como la motivación.En el capítulo 2 se discute las diferentes connotaciones que tienen los hilos de ejecución, desde el punto de vista del microprocesador, del sistema operativo y del intérprete. En este mismo capítulo se abordará el tema de la transportabilidad de aplicaciones gráficas utilizando *Python* como lenguaje interpretado y el *toolkit WxWindows*. En el capítulo 3 se verán en detalle los puntos centrales para el desarrollo de dos aplicaciones bajo el mismo caso de estudio asociado con un sistema para el análisis de señales *Doppler* ultrasónicas. Ambas estrategias cumplen con los requerimientos de una aplicación de TRS. La diferencia entre ambas estriba en la manera de controlar el despliegue del espectro, si se le trata como un hilo más o se controla en el *"loop"* del manejo de los eventos, manipulando el despliegue mediante un planificador. En el capítulo 4 se mostrarán todos los resultados obtenidos por las dos aplicaciones en los sistemas *Linux* y *Mac Os X*. En el capítulo 5 se reportarán las conclusiones del trabajo.

Capítulo 2

Sincronización de procesos.

En el presente capítulo definirá los *hilos* de *CPU* (*multithreading*), los *hilos* de *SO* y el *hilo principal* del intérprete. Se abordará la manera en que el intérprete de *Python*, implementa y controla los hilos de *SO*. Además se hablará de la interacción entre los hilos y los eventos de los eventos gráficos.

2.1. Hilos de CPU.

Para mejorar los desempeños de los microprocesadores, sus fabricantes utilizan técnicas como el paralelismo a nivel de instrucción, y el paralelismo de *multithreading* en el mismo *"chip"*. Dichas técnicas involucran directamente el *pipeline* y la memoria *cache* del microprocesador.

El *pipeline* es un paradigma, mediante el cual los subsistema de direccionamiento y de ejecución, se pueden ir "adelantando instrucciones". Esto es, al tiempo que el subsistema de ejecución del micro está trabajando, el susbsistema de direccionamiento puede seguir trayendo de la memoria principal las instrucciones siguientes, e irlas poniendo en alguna de las etapas del *pipeline*. En la Figura 2.1 se muestra de modo esquemático, la paralelización intrínseca del *pipeline* de instrucciones. La memoria *cache* es un recurso muy utilizado en la arquitectura de los microprocesadores. mediante el cual, se agiliza el intercambio

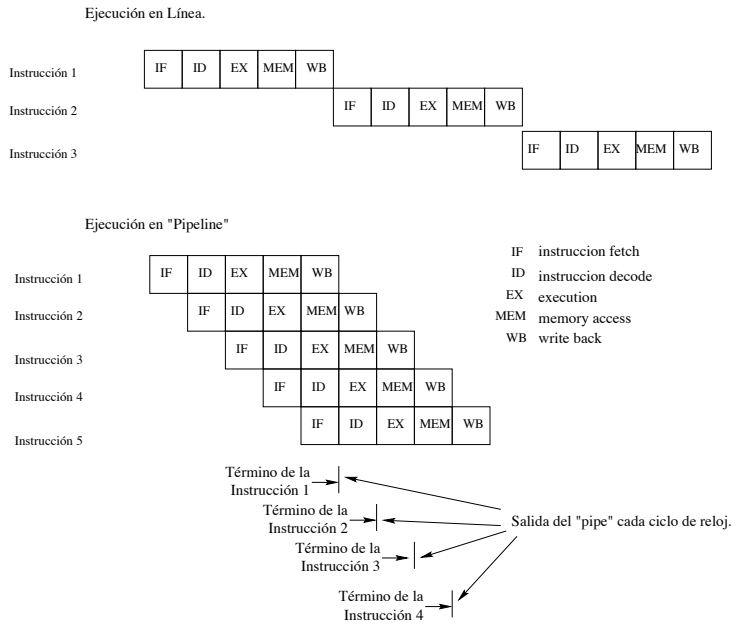


Figura 2.1: **Pipeline de instrucciones. En la parte inferior se aprecia que por cada ciclo de reloj, se obtiene un ciclo completo del microprocesador (IF, ID, EX, MEM, WB).**

entre memoria y microprocesador. Es memoria de acceso rápido y de modo asociativo, por tanto de mayor costo que la memoria convencional. En ella se van guardando los contenidos de la memoria principal más recientemente utilizados. Debido su alto costo, el tamaño de dichas memorias van de los 16 KB a los 64 KB. Por tener una eficiencia aceptable, es común encontrarla en varias arquitecturas de microprocesadores, incluso en varios niveles, denominados como L1 (nivel 1) el principal, L2 (nivel 2) y subsecuentes niveles de caches secundarias (L3...), en donde cada nivel secundario, es de mayor tamaño, (van de los 512 MB a 1 GB) y su tiempo de acceso más lento y por tanto son más económicas.

Al estar en ejecución el sistema, puede llagar el momento en que una referencia de memoria pierda sus *cache*s de nivel 1 y 2, ocasionándose una "espera" hasta que la "palabra" requerida y su linea asociada a su *cache* sean cargadas desde la memoria principal hacia las memorias cache. A lo anterior se le conoce como rompimiento del *pipeline*[Tanenbaum, 2006]. Para solucionar el problema anterior, se propone que el propio microprocesador enmascare este tipo de ruptura del *pipeline*, proporcionándole múltiples

hilos de control. A esto se le denomina "on-chip *multithreading*". En otras palabras si el hilo 1 se atasca, el microprocesador tiene la manera de seguir la ejecución con un segundo hilo, consiguiéndose que el *CPU* se mantenga lo más ocupadamente posible.

Concretamente, el *Pentium IV* de 3 GHz de *Intel*, tiene lo que se denomina *hyperthreading*, *HiperThreading*, el cual consiste en dos unidades para manejar sendos hilos de ejecución.

Lo anterior aunado con la evolución de la tecnología de integración, ha dado pié a tener distintos *"cores"* o subsistemas como: *pipeline*s, *ALU*s y *CPU*s en un mismo chip. A esto se le denomina *múltiple "core"*. Se le denomina *"dual core"* al sistema que en una misma oblea tenga dos microprocesadores, compartiendo los demás componentes del sistema como: memoria, cache, etc. Esta tendencia ha tenido éxito debido a que resuelve problemas de disipación de calor. Por ejemplo los procesadores *Pentium IV* de 4 GHz. necesitan sistemas especializados de enfriamiento. Por tanto se ha demostrado que es más efectivo tener en un misma oblea o *Core* dos microprocesadores de hasta 3 GHz.

2.2. Hilos en el sistema operativo.

El proceso es una instancia de un programa junto con la información de su estado (memoria, registros, contador de programa, status de entrada y salida, etc.) [Tanenbaum, 2006, Process Wikipedia, 2006]. Los procesos son controlados por el *SO* y éste proporciona los medios para la comunicación entre ellos, como: *pipes*, *FIFOs*, etc. [Stevens, 1992]. Todo proceso tiene intrínsecamente asociado a el un *hilo* (*thread*). Dicho hilo se le puede ver como la hebra que va hilvanando los caminos que tiene un proceso al irse decodificando sus instrucciones en el *CPU*. Esta idea pretende ser esquematizada en la Figura 2.2, en donde se aprecia dicho *thread* con un cambio de contexto para atender una función o subprograma. La idea principal es que, el mismo *thread* hilvane todas las instrucciones que conforman al proceso con todo y el salto a función.

Uno de los desafíos del *SO* es poder controlar varios procesos de forma "concurren-

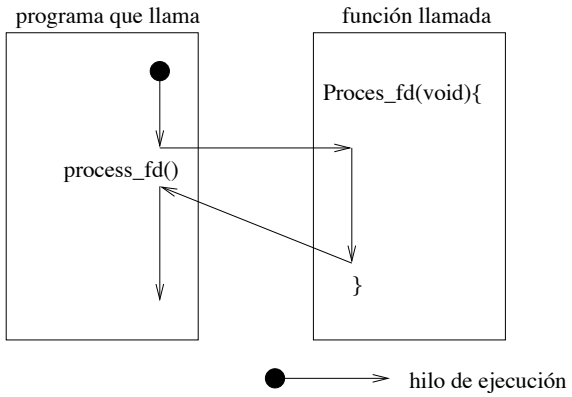


Figura 2.2: **Esquema del *hilo* principal de un proceso, resaltándose una llamada a una función.**

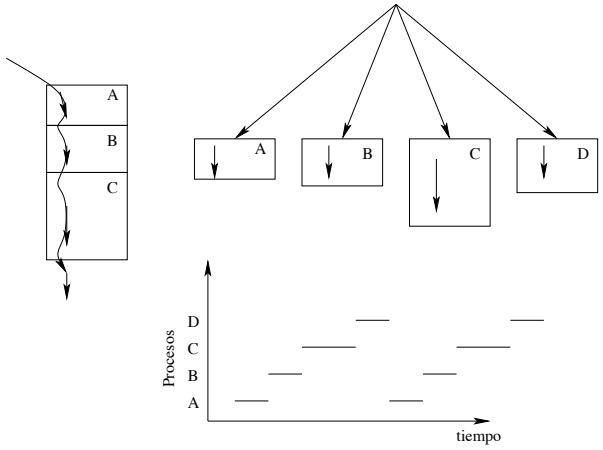


Figura 2.3: **Esquema del manejo de múltiples procesos por el *SO*.**

te", mediante el componente fundamental de éste, el planificador o *scheduler*, como se esquematiza en la Figura 2.3. Internamente el *SO*, le asigna tres tipos de estados a los procesos: corriendo, listo y bloqueado [Tanenbaum, 1992] como se aprecia en la Figura 2.4. Por tanto de todos los procesos que estén siendo atendidos por el *SO*, solo uno puede estar activo en algún instante de tiempo.

Es posible, que a nivel de usuario, dentro de una misma instancia de algún proceso, existan asociados a el varios hilos, dando lugar al paradigma de programación conocido como hilos múltiples. Dichos hilos comparten el estado del proceso el cual los generó (memoria, registros, etc.) La idea general es que el programa principal "lance" uno o varios "procesos" para que estos se ejecuten en sendos hilos, como se esquematiza en la Figu-

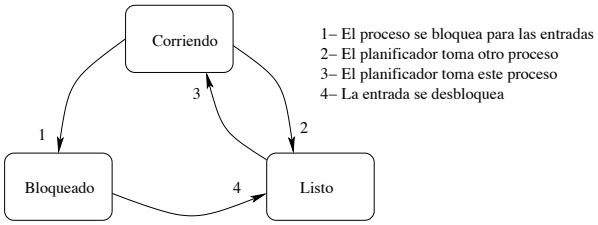


Figura 2.4: **Estados de un proceso.**

ra 2.5.

Depende del diseño del *kernel*, si éste tiene o no injerencia sobre los hilos creados. Si el *kernel* no tiene control sobre los hilos, a éstos se les denomina procesos ligeros (LWP, Light Weight Process) o *hebra* como se le denomina en la jerga de de algunos sistemas operativos. Si el *SO* tiene algún tipo de control sobre éstos, se les llama: hilos de *kernel*.

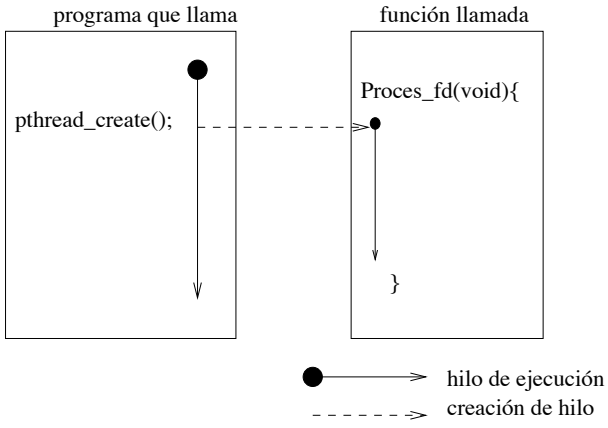


Figura 2.5: **Esquema de un hilo lanzado por un proceso.**

Existen pros y contras en cada uno de los hilos mencionados anteriormente, El más importante es cuando algún hilo hace alguna llamada de *IO* y cause un bloqueo. El manejo de tal situación dependerá de la implementación de los hilos. Si se tratan de *hebras* el bloqueo de una afectará a todas las demás que estén activas en el proceso dado. Debido a que aquellas están controladas por un planificador a nivel de la aplicación. Con los hilos de *kernel*, se puede cambiar de contexto, haciendo que el *kernel* se "enganche" al hilo activo. Lo anterior da la solución para que no haya hilos que bloqueen a los demás, por que el *kernel* con ayuda del planificador soluciona los bloqueos debidos al *IO*.

Esto se soluciona en el planificador con la inversión de prioridades [Tanenbaum, 1992, Hristu-Varsakelis et al., 2005].

El inconveniente que presentan los hilos, es que no solo hay que crearlos, además hay que controlarlos. Además no es fácil detectarlos al depurar el código, ó hacer el perfil de desempeño (profiler). Lee en [Lee, 2006b], muestra de manera lógica por que no se deben dar soluciones a la ligera con hilos. Ousterhout en [Ousterhout, 1996] afirma lo anterior y su tesis fundamental es que es mas fácil programar bajo el paradigma de eventos. Los medios ambientes gráficos están basados en eventos. Ousterhout es el creador de *Tcl-Tk* que es un lenguaje interpretado muy usado junto con *Tk* para la creación de interfaces gráficas de usuario. De ahí que Ousterhout comulgue con los eventos. Sin embargo menciona que para hacer cómputo científico, es en donde se pueden utilizar los hilos para tener mejores desempeños. En otro artículo hecho por Lee [Lee, 2006b] se propone, para que sea menos dolorosa y en sus propias palabras "insane" la programación de hilos, hay que basarse en *patrones de software*.

Los problemas más comunes que se presentan al programar con hilos son: La competencia de recursos, y el bloqueo al hacer alguna instrucción de *IO*. Para solucionar el primer problema, las bibliotecas para la creación y manejo de hilos, tienen instrucciones para sincronizar, y hacer exclusiones mutuas, aparte de garantizar la "atomicidad" de las instrucciones. Funciones como `setlock, getlock,` son fundamentales para garantizar la exclusión mutua.

Por último mencionaremos los siguientes sistemas operativos así como el tipo de hilos que implementan. Win32 implementa hebras, SunOS 4.x implementa procesos ligeros (light weight, o LWP) como fibras conocidos como hilos verdes (green threads). SunOS 5.x y mas recientes, NetBSD 2.x y DragonFly BSD implementan procesos ligeros como hilos. Los hilos son las unidades mas ligeras del planificador del kernel. Éstos son *pre-emptive* si el planificador es *pre-emptive*. Los hilos no tienen recursos propios excepto el "stack" y una copia de los registros incluido el contador de programa. [Thread Wikipedia, 2006]. *Linux* y *Mac Os X* tienen hilos de *kernel*. En el caso de los sistemas operativos que tienen

hilos de *kernel*, también pueden tener la implementación de hebras, o sea, son híbridos.

Con lo anterior hemos dado una semblanza del paradigma de hilos múltiples. Se parte del hecho que para programarlos, el *SO* proporciona las bibliotecas para el desarrollo de aplicaciones (generalmente hechas en lenguaje "C"). En el presente trabajo se quiere tener la perspectiva de programación desde un intérprete, para lo cual es necesario entender como éste trabaja con los hilos, como se verá a continuación.

2.3. Hilos del intérprete.

Internamente el intérprete tiene una instancia de su "máquina virtual" en algún *SO*. Dicha "máquina virtual" ejecutará un tipo de código que el intérprete haya generado. Por ejemplo en la especificación de *Java*, se tiene muy claro el concepto de la "máquina virtual". Pero esto no restringe a que se pueda generalizar el concepto de "máquina virtual" en otros lenguajes interpretados como *Python*.

Para que *Python* pueda manejar *multithreading*, necesita que el intérprete "enganche" al hilo principal del proceso en cuestión. Esto lo hace mediante el *GIL* ("Global Interpreter Lock"). Esto es que el hilo principal del programa, debe ser enganchado por el intérprete antes de que el intérprete tenga acceso a los objetos de *Python* [Python API, 2006]. Esto da base para el manejo de las medidas de perfiles *"profiling"* incluidos los *threads* en *Python*.

Python maneja en su semántica el concepto de módulo [Cárdenas Flores, 1996]. El módulo principal, para el manejo de los *threads* es el módulo `thread`. Existe otro módulo, el `threading`, que está basado en el manejo que hace *Java* de los *threads*. En dicho módulo están implementadas clases como: `Conditions` y `Semaphors`. También existe el módulo `Queue` en donde está implementada la clase `Queue`, la cual sirve para sincronizar los diferentes *threads* que quieran tener acceso a alguna instancia de `Queue` utilizando los objetos `lock`, éstos objetos son los encargados de sincronizar los *threads* que compartan un área crítica, asignando el siguiente *thread* en ejecución. Esto es, que los objetos `lock`

hacen la exclusión mutua de los *threads*.

Para la implementación del módulo `thread` se utiliza el tipo de hilos que el *SO* otorgue. Por ejemplo los hilos que en *Linux* y en *Mac Os X* utilizan son los `pthreads`, o hilos *POSIX* (Portable Operating System Interface for uniX) [Robbins and Robbins, 2003].

2.4. Hilos y eventos.

Hasta aquí hemos hablado de las connotaciones de los *hilos* en los sistemas de cómputo. Éstos se restringen al procesador y el *SO*. Para la construcción de interfaces gráficas de usuario, como se mencionó al principio de este trabajo, el paradigma de programación es el de *eventos*. La interacción de los *hilos* con los *eventos* de una interfaz gráfica depende del *toolkit* en el que se esté trabajando. Así por ejemplo en *GTK*, con su módulo para *Python*, *PyGTK*, no hay manera de interacción con los *hilos*. Los eventos se manejan concurrentemente y los controla el mismo *toolkit*, mediante la declaración de áreas críticas. En cambio *WxWindows*, en su módulo para *Python*, conviven sin ningún problema sus eventos con los hilos declarados mediante el módulo `thread`, Tal es su interrelación, que se puede hacer herencia múltiple con una *clase* de algún elemento gráfico y la clase `Thread`.

En el presente trabajo se utilizó *WxWindows* como *toolkit* de desarrollo para obtener dos casos o programas que correrán en dos diferentes *SO*s: *Linux* y *Mac Os X*.

En el siguiente capítulo se describirán los pormenores para la implementación del sistema que evaluará los desempeños del manejo de los *threads* en los sistemas *Linux* y *Mac Os X*.

Capítulo 3

Desarrollo del sistema.

3.1. Generalidades del sistema.

Para hacer análisis y diseño orientado a objetos [Booch, 1994], se seguirá el esquema hecho en la Figura 1.2, considerándose a cada elemento de dicha Figura como un objeto. La idea principal es obtener un desacoplamiento entre los procesos (hilos) de Captura, Procesamiento y Despliegue mediante el intercalamiento de dos *FIFOs*. El hilo Captura será el encargado de la comunicación con el dispositivo de audio, mediante el demonio *esd* (Enlightened Sound Daemon), leyendo N muestras correspondientes a la ventana w las cuales almacenará en el primer *Queue* concurrente. El hilo encargado de hacer el procesamiento de la señal capturada, leerá los datos del primer *Queue* concurrente, para hacer el cálculo correspondiente de la ventana de tiempo w depositando su resultado en el segundo *Queue* concurrente. El tercer objeto, el Despliegue, leerá los datos ya procesados del segundo *Queue* concurrente, y los adecuará para su visualización.

Los hilos involucrados para cada proceso son asíncronos, y los elementos encargados de sincronizar los procesos involucrados son los *Queues* concurrentes. Al encargase los *Queues* de la sincronización de los hilos involucrados, estos sólo tienen que hacer lecturas o escrituras a dichos *Queues* sin que se involucre ningún tipo de código ni de objetos para hacer la sincronización, (como Locks, semaphores, etc). Obteniéndose así un código

sencillo de seguir.

Debe tomarse en cuenta que el proceso de despliegue está supeditado al ambiente de ventanas en el que se esté trabajando, y que el paradigma de programación de dichos ambientes es el de eventos. Todo lo que ocurre en el ambiente de ventanas es un evento. El movimiento del ratón, la acción de algún botón del mismo ratón, la activación de alguna ventana, o simplemente si el sistema esté sin hacer nada `EVT_IDLE`, etc. Debe entonces ponerse atención a la interacción entre los hilos de la `Captura` y el `Procesamiento` con los eventos para el despliegue. Los IDEs y Toolkits para el desarrollo de aplicaciones en algún ambiente de ventanas, proporcionan la manera de hacerse la interacción entre los procesos (hilos) y los eventos gráficos del sistema.

La aplicación que se utiliza como caso de estudio, es un analizador de espectro para un sistema de *Doppler* ultrasónico utilizado en el diagnóstico del flujo sanguíneo. Hay dos tipos de sistemas para la detección de flujo sanguíneo, los de onda continua y los de onda pulsada, y en cada uno de estos tipos, existen desarrollos para la detección de flujo bidireccional [Fuentes et al., 2006, García et al., 2006]. Un esquema general para un sistema *Doppler* de onda continua se puede apreciar en la Figura 1.3.

El detector de flujo de onda continua, tiene un transductor que emite el haz ultrasónico (de 4 a 8 MHz.) de forma continua, y otro que recibe la señal de *"backscattering"* producida por la interacción del ultrasonido con el tejido hemático. A la diferencia de frecuencia entre las dos señales se le llama frecuencia *Doppler*. Si se considera una sola partícula con la interacción de un haz ultrasónico a un ángulo θ, la diferencia de frecuencia está dada por:

$$\Delta f = \frac{2|\vec{v}|\cos\theta}{c} f_0, \tag{3.1}$$

en dónde: v es la velocidad de la partícula, θ es el ángulo entre el haz ultrasónico y la trayectoria de la partícula, c es la velocidad del sonido en el medio de propagación dado y f_0 es frecuencia del haz incidente [Jensen, 1996, Evans and McDicken, 2000]. Como puede apreciarse en la ecuación de arriba, la diferencia en frecuencia es proporcional

a la velocidad de la partícula en cuestión. La sangre es un conglomerado de pequeñas partículas, eritrocitos, leucocitos y plaquetas, siendo las de mayor número los eritrocitos. Considerándose que la sangre está constituida por un 45 % de eritrocitos (hematocrito) [Guyton, 1987], la mayor contribución a la señal de *"backscattering"* es debida éstos. Se habla entonces que la distribución de las frecuencias *Doppler* captadas por el equipo, corresponde a la distribución de las velocidades a lo largo del diámetro de la arteria en estudio.

Si substituimos en la Ecuación 3.1 de arriba los valores típicos de: velocidad de la sangre ($v = 0{,}5$ ms), frecuencia del haz ultrasónico($f_0 = 8$ MHz), un ángulo $\theta = 45°$, y la velocidad del ultrasonido en el cuerpo humano ($c = 1540$ ms), obtenemos que la frecuencia *Doppler* es de 3673.285 Hz. y que está en el rango audible. Es por esto que los equipos de diagnóstico angiológico, tienen salida para altavoces, utilizada para la auscultación. El audio de esta señal es artificial, y es debido a la extraordinaria combinación de los valores físicos de la ecuación 3.1.

3.2. Caso de estudio.

En este capítulo se describen el análisis y diseño de las dos aplicaciones para el casos de estudio que se considera en el presente trabajo. El fin de ambos sistemas es evaluar el desempeño de los *threads* en los dos diferentes sistemas operativos involucrados, *Linux* y *Mac Os X*. El objetivo de cada sistema es desplegar el espectro de la señal *Doppler*. Mitra en [Mitra, 2001] define al espectrograma como despliegue de la magnitud de la transformada de *Fourier* de corta duración ($STFT$). La $STFT$ está dada por:

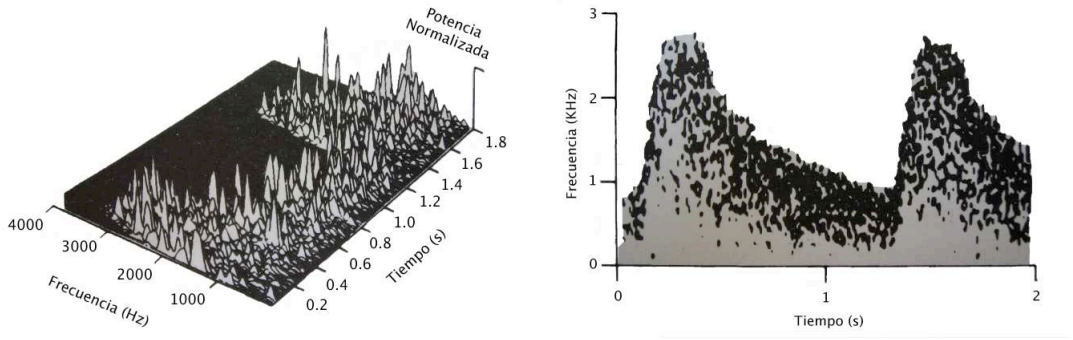
$$X_{STFT}[k,n] = \sum_{m=-\infty}^{\infty} x[n-m]w[m]e^{-j\omega_k m}, \quad \text{con} \quad \omega_k = \frac{2\pi}{R}k \tag{3.2}$$

en donde $w[n]$ es alguna función de ventana a seleccionar (Rectangular, Hamming, Hann, Kaiser etc) y R es el tamaño de la transformada de *Fourier*. Lo que se busca para fines

prácticos, es que dicha ventana extraiga una porción finita de la secuencia $x[n]$ tal que sus características espectrales sean aproximadamente estacionarias en el rango de tiempo determinado por la ventana $w[n]$. Nótese que si $w[n] = 1$ la transformada arriba descrita se reduce a la transformada de *Fourier* de tiempo discreto ($DTFT$) de la secuencia $x[n]$.

De la definición dada en el Capítulo 1, el tamaño de la ventana $w[n]$, para el cálculo de la magnitud de $STFT$ debe ser de 10-20 ms. para señales *Doppler* ultrasónicas.

El resultado total del sistema se visualiza en la Figura 3.1, en donde los valores de cada espectrograma o línea espectral se representan como alturas sobre el plano *tiempo-frecuencia* o como intensidades de grises (o de alguna escala de color)



(a) Representación en 3D (b) Escala de grises o de colores

Figura 3.1: **Representación del espectro: a) En 3D. b) Escalas de grises o de colores.**

De la idea fundamental esquematizada en la Figura 1.2, se desarrollan dos casos de

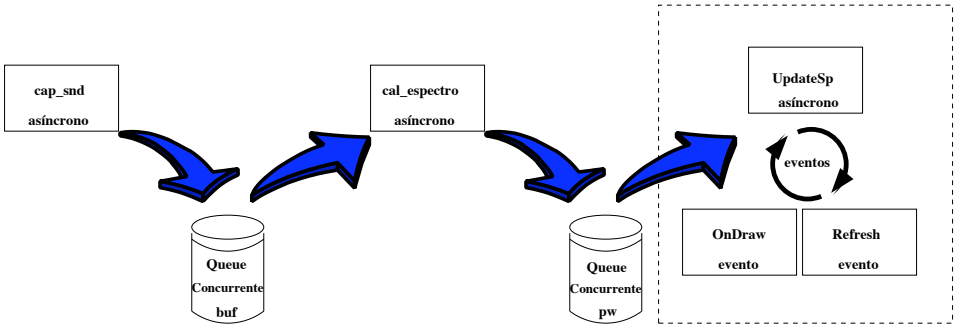


Figura 3.2: ***Caso 1*: Esquema en el cual todos son procesos son asíncronos.**

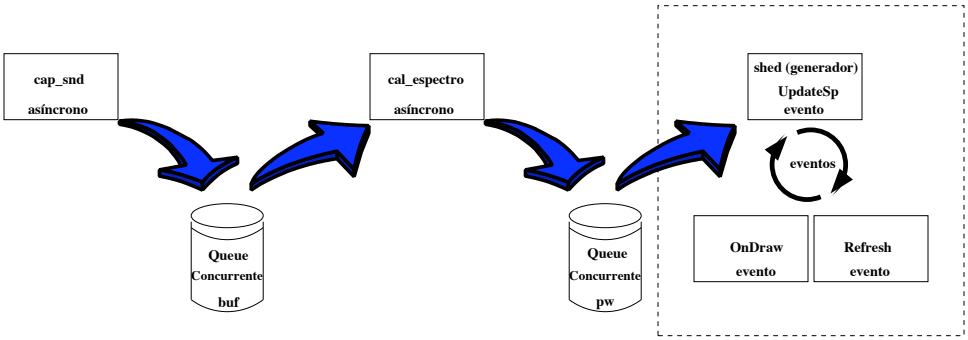


Figura 3.3: ***Caso 2:*** **Utilizando generadores (iteradores)**

estudio con las características siguientes: Caso 1, Figura 3.2, todos los procesos de captura, cálculo y despliegue son asíncronos, y se sincronizan por medio de los *Queues* concurrentes. Caso 2, Figura 3.3, solo el despliegue se hace sincrónico mediante un planificador.

A continuación se darán los pormenores del análisis y diseño para ambos casos.

3.3. Descripción general del sistema.

A continuación se explicará cada uno de las etapas fundamentales que componen el sistema: captura, cálculo y despliegue de la señal *Doppler* ultrasónica.

3.3.1. Captura de la señal.

Como se aprecia en la Figura 1.3, la señal que sale del sistema *Doppler* de onda continua se introduce a la computadora por la entrada de audio. Por tanto lo primero que hay que hacer es, leer el dispositivo de audio de la computadora en *tiempo real*. Para el control del dispositivo de audio se emplea el *esd*[1] que es muy popular en las distribuciones de *Linux* tales como *RedHat* y *Debian*, y que además se le encuentra en *Mac Os X*. *esd* es un demonio que facilita la comunicación entre la computadora y el dispositivo de audio, permitiendo la lectura y escritura (captura y reproducción) de la señal de audio. Además da acceso al dispositivo vía red mediante *sockets* como se muestra en la Figura 3.4.

[1] *esd* proviene de las siglas en inglés *Enlightenment Sound Daemon.*

El módulo de *Python* que involucra el *API* de *esd* se le denomina *PyESD*. En este módulo, aparte de ser la envolvente de la componente *esd*, implementa varias *clases*.

Para el sistema se diseñó la *clase* `Recorder`. El diagrama de clases [Booch et al., 1999] de `Recorder` se puede ver en la Figura 3.5, en donde se aprecia que dicha clase es heredera de la *clase* `ServerConnection`, que es parte de *PyESD*. En `Recorder` se implementaron dos funciones miembro: `read()` para la lectura del dispositivo de audio y `rep(cad,rate)` para la reproducción de la cadena `cad`. Dicha cadena de palabras puede ser de 8 o 16 bits y tener una frecuencia de muestreo dada por `rate`, según se haya configurado en el constructor de `Recorder`. Como se puede ver en la Figura 3.5 la función `rep` está relacionada con `ServerConection` que a su vez depende de la clase `Sample` que están definidas en *PyESD*.

Los parámetros que inician al *objeto* `Recorder` son:

format Formato para la captura. El formato estándar es: `ESD_RECORD | ESD_MONO | ESD_BITS8`, que indica: que se va a grabar (`ESD_RECORD`) en un solo canal (`ESD_MONO`) palabras de 8 bits (`ESD_BITS8`). Todos estos valores están definidos en la *API* de *esd* y éstos se encadenan con el "or" (`|`) como normalmente se hace en lenguaje *C*.

rate Es la frecuencia a la cual se muestrea la señal.

host Es la computadora que tiene el dispositivo de audio (si se está trabajando en un esquema de red).

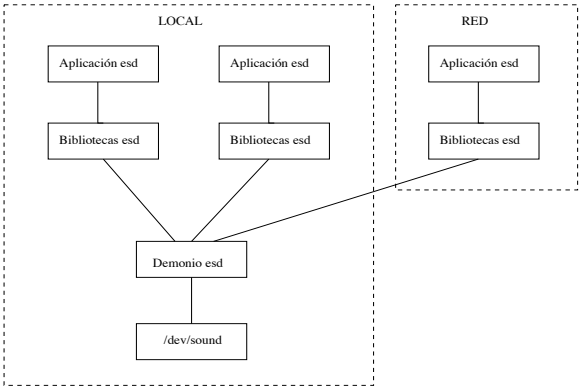


Figura 3.4: **Estructura de *esd***

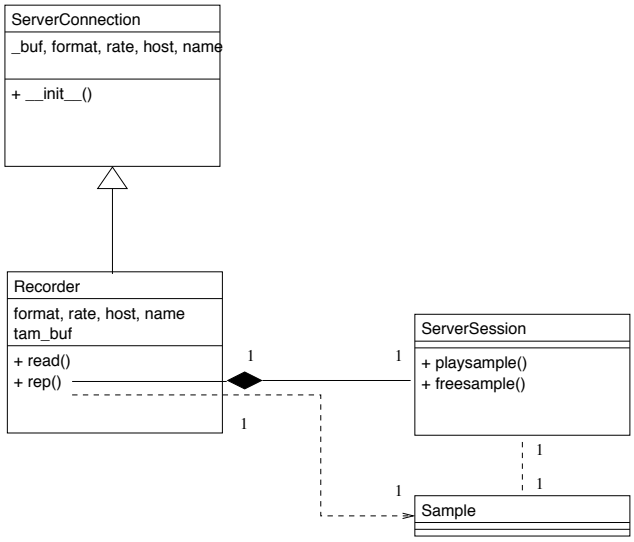


Figura 3.5: **Clase** `Recorder`

name Es el nombre del *stream*. Dicho *stream* al tener acceso al servicio de *esd* para reproducir o grabar, se identifica por su nombre `name`.

tam_buf Es el tamaño de la ventana.

El algoritmo que se usa en la función miembro `read()` del *objeto* `Recorder` para la captura de la señal es el siguiente:

```
def read():
    lleno = 0
    tam = 0
    buf = ""
    while not lleno:
        tmp = read(_fd,tam_buf-tam)
        buf = buf + tmp
        tam = tam + len(tmp)
        if tam  == tam_buf:
            lleno = 1
    return buf
```

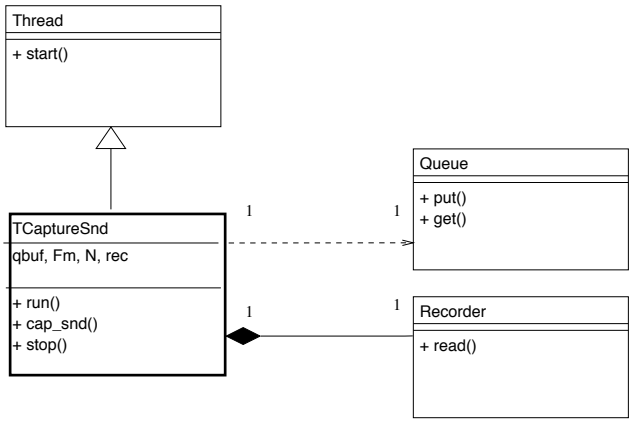


Figura 3.6: **Diagrama de clases para la captura de sonido.**

Lo que hace este algoritmo es leer una ventana de tamaño `tam_buf`. La función `read` que se emplea es la definida en *UNIX* en `unistd.h` [Stevens, 1992], cuyo descriptor de archivo `_fd` corresponde al dispositivo de audio el cual es proporcionado por `ServerConection`. No es una llamada recursiva (para eso, se haría con `self.read()`). Dicha función se incorpora a *Python* mediante el módulo `os`, el cual se inicia debidamente dependiendo del sistema operativo (*Linux*, *Windows*, *Mac Os X* etc.). Las líneas subsecuentes a la función `read` son para verificar que se leyeron los N bytes, correspondientes al tamaño de la ventana, representados en la variable `tam_buf`.

La captura de sonido es de forma *asíncrona*, para los dos casos. Por tanto se hace uso del recurso que *Python* da mediante la clase `Thread`, auxiliándose de la clase `Queue` para guardar la captura de la señal de audio y sincronizarse con los demás hilos de ejecución del sistema. Se diseñó la *clase activa* `TCaptureSnd`, cuyo diagrama de clases se puede ver en la Figura 3.6. Dicha clase está relacionada en forma de composición con `Recorder` y depende de `Queue`. La función en donde se involucra el algoritmo de captura descrito en esta sección es `cap_snd()`. De esta manera, mediante el parámetro de inicialización `N` de la clase `TCaptureSnd`, se le indica el tamaño de la ventana para el cálculo de la *STFT* en la Ecuación 3.2.

3.3.2. Cálculo del periodograma.

Para cada muestra guardada en el *Queue* de entrada $x[n]$, se necesita estimar su espectro de potencia $\hat{\mathcal{P}_{xx}}$ dado por la siguiente ecuación:

$$\hat{\mathcal{P}}_{xx}[k] = \frac{1}{CN} \left|\Gamma[k]\right|^2, \tag{3.3}$$

en donde

$$\begin{aligned} \Gamma[k] &= \sum_{k=0}^{R-1} x[n]w[n]e^{-j\omega_k n} \quad \text{con} \quad \omega_k = \frac{2\pi}{R}k \\ C &= \frac{1}{N}\sum_{n=0}^{N-1} |w[n]|^2 \end{aligned} \tag{3.4}$$

En nuestro caso, la secuencia $w[n]$ es una ventana rectangular, por tanto de la Ecuación 3.4 $C = 1$. Cuando la secuencia de la $w[n]$ es rectangular, la cantidad de la expresión 3.3, se le define como *periodograma*, y *periodograma modificado* cuando dicha secuencia es cualesquiera otra.

En la práctica el *periodograma* $\Gamma[k]$ es evaluado a R muestras discretas igualmente espaciadas, ($0 \geq k \geq R-1$, ω_k Ecuación 3.4). La secuencia $x[n]$ consta de N muestras discretas, si se hace que $R > N$, se obtiene una malla mas fina de muestras del *periodograma*. El periodograma $\hat{\mathcal{P}}_{xx}[k]$ expresado en la Ecuación 3.3, en realidad es una estimación del periodograma $\mathcal{P}_{xx}$ que está definido por (ver [Mitra, 2001, Proakis and Manolakis, 1998]):

$$\mathcal{P}_{xx}(w) = \sum_{\ell=-\infty}^{\infty} \phi_{xx}[\ell]e^{-j\omega\ell}, \tag{3.5}$$

en donde $\phi_{xx}[\ell]$ es la autocorrelación de la secuencia $x[n]$ definida por:

$$\phi_{xx}(\ell) = \mathcal{E}(x[n-\ell]x^*[n]). \tag{3.6}$$

En donde $\mathcal{E}(\cdot)$ denota el valor esperado. Como se define en [Proakis and Manolakis, 1998],

existen dos métodos de aproximación al *periodograma*, uno directo que consiste en calcular la transformada de *Fourier* discreta (DFT) de la secuencia truncada por la ventana rectangular $w[k]$ (Ecuacion 3.3), y el método indirecto, que consiste en primero calcular la secuencia de autocorrelación de la secuencia[2] $x[n]$.

Aprovechando la existencia de la implementación del algoritmo rápido de la transformada de *Fourier*, (FFT) que está en el módulo de *Python* denominado `scipy`, utilizaremos el método directo para el cálculo del *periodograma*. El módulo `scipy` básicamente incorpora a *Python* tipos numéricos reales y complejos con una precisión de hasta 64 bits, equivalente en lenguaje C a un *double*, que pueden estar arreglados en listas o vectores o matrices bidimensionales y tridimensionales. Además en la definición de los arreglos, existen operadores de deslizamiento, para tener acceso no sólo a una entrada, si no a un rango de entradas. En el módulo `scipy` además existen implementaciones de algoritmos para el cálculo de funciones especiales como las de *Bessel, Gamma* , etc.

Para calcular el espectro de la ventana en cuestión se lee del *Queue* que contiene las capturas hechas por el objeto de la clase `TCaptureSnd`, y se procede a calcular el *periodograma* de la muestra en turno, Ecuación 3.4, mediante el siguiente código:

```
buf = fromstring(self.qbuf.get(),UnsignedInt8)
pw = pow( abs(fft(buf)) ,2 ) / len(buf)
```

En donde la primera línea indica que se extrae del buffer `qbuf` y se pasa a la variable `buf` con formato de caracter sin signo, mediante el procedimiento `fromstring`. En la siguiente línea se calcula el *periodograma* en cuestión.

Para tener una mejor visualización del *periodograma* arriba calculado, es de uso común hacer una normalización en decibeles. Dicha normalización se hace de la siguiente manera:

```
pw = 20 * log10( pw / max(pw))
for i in xrange( len(pw)):
```

[2]A los métodos planteados en este capítulo se les denominan también métodos *no paramétricos*. Existen otros métodos llamados los paramétricos ver [Mitra, 2001, Oppenheim et al., 1999, Proakis and Manolakis, 1998]

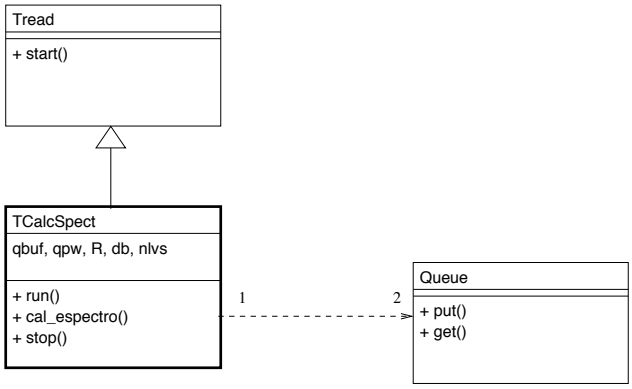


Figura 3.7: **Diagrama de clases para el cálculo del espectro.**

```
    if pw[i] <= -db:
       pw[i]   = -db
pw = pw / db * nlvs
```

En la primera línea, el cálculo de los decibeles se hace multiplicando por 20, debido a que el periodograma viene del cálculo de una magnitud al cuadrado, ver Ecuación 3.3. Después se hace una discriminación de los niveles menores a `-db`. La última línea normaliza los valores de `pw` a el rango $\{0, 1, 2 \ldots nlvs - 1\}$, que corresponden a el número de niveles de la visualización. En este caso el valor de `nlvs` es de 256.

Por último, resta guardar el cálculo hecho del *peridograma* en el *Queue* respectivo con la instrucción `qpw.put(pw)`.

Como en el caso de la captura de la señal de audio, el cálculo del espectro para ambos casos, también se hace de forma asíncrona. En la Figura 3.7 se muestra el diagrama de clases para el cálculo del espectro. La *clase activa* `TCalcSpect` depende de la clase `Queue`. La cardinalidad que se muestra es de 2, por que se cuenta con un `Queue` para la captura y otro para almacenar los vectores del cálculo del *periodograma*.

3.3.3. Construcción y despliegue del espectrograma.

Hasta ahora hemos capturado secuencias $x[n]$ de tamaño N y calculado el *periodograma* de tamaño R, en donde $R \geq N$, de los segmentos de la secuencia $x[n]$. Los valores del

periodograma (`pw[i]`) se han normalizado al rango $\{0, 1, 2 \ldots 255\}$. Se requiere representar dichos valores a sus correspondientes en niveles de grises o de colores.

Para desplegar el espectro de la señal *Doppler* se utiliza *OpenGL* [Neider et al., 1993], que tiene su módulo para *Python* llamado *pyOpenGL*. *OpenGL*, es una biblioteca para la graficación por computadora desarrollada por *Silicon Graphics*. Los sistemas *GNU* basados en *Linux*, cuentan con *MESA*, que es una implementación *libre* de *OpenGL*.

Para definir la escala de grises en *OpenGL*, se asignan los i-ésimos componentes de color *RGB*, a un mismo i, como se hace en el siguiente código.

```
grises = ones(nlvs*4,UnsignedInt8)
for i in xrange(nlvs):
   grises[i*4  ] = i  # Rojo
   grises[i*4+1] = i  # Verde
   grises[i*4+2] = i  # Azul
   grises[i*4+3] = 0  # Alfa
```

El último componente denominado *alfa* es de transparencia, que en este caso no se ocupa. La escala que se obtiene se muestra en la Figura 3.8a.

La escala cromática en *OpenGL*, se genera de la siguiente manera:

```
esc_crom = ones(nlvs*4,UnsignedInt8)
```

(a) Escala de grises

(b) Escala de colores

Figura 3.8: **Escalas para el despliegue.**

```
cte = 2.0 * pi / (nlvs)
for i in xrange(nlvs):
   ti = cte * i
   if i <= nlvs/2:
      esc_crom[i*4+2] = 127*(1.0 + cos(ti))
      esc_crom[i*4+1] = 127*(1.0 + cos(ti+pi))
   else:
      esc_crom[i*4  ] = 127*(1.0 + sin(ti+pi))
      esc_crom[i*4+1] = 127*(1.0 + cos(ti+pi))
```

La escala que se obtiene se muestra en la Figura 3.8b.

El mapeo de los valores del *periodograma* a alguna de las escalas que arriba se crearon, se hace mediante la asignación de la referencia `escala`, como se aprecia en el siguiente código:

```
lvs = pw.astype(Numeric.UnsignedInt8)
for i in xrange(len(pw)/4):
      l[i*4]   = escala[lvs[i]][0] # Rojo
      l[i*4+1] = escala[lvs[i]][1] # Verde
      l[i*4+2] = escala[lvs[i]][2] # Azul
```

Con la primera línea se asegura que los valores normalizados del i-ésimo *periodograma* `pw` estén representados como caracteres sin signo en la variable `lvs`. El arreglo `l[]` está en formato *RGBA* y el valor de sus entradas corresponden a los colores en el que se va a desplegar la línea `l[]`, resultado de mapear los valores de `lvs` a la `escala` que está en formato *RGBA*.

Las funciones encargadas de hacer las escalas de grises y de colores son: `CreateGraySc` y `CreateColorSc` respectivamente. Y la función encargada de hacer el mapeo de los valores calculados del espectro a su respectiva escala de grises o colores, es `UpdateSP()`. En las Figuras 3.10 y 3.11 se aprecian los diagramas de clases para los dos casos en que van a

trabajar las funciones que aquí se describen.

3.3.4. Despliegue del espectrograma.

Yá que se ha calculado el *periodograma*, ahora procederemos a desplegar el espectrograma.

Las funciones que inician el despliegue en *OpenGL* son las siguientes:

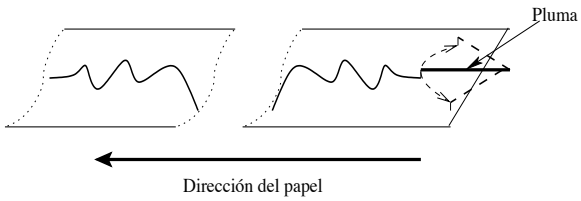
```
glMatrixMode(GL_PROJECTION)
glLoadIdentity()
glOrtho(0, w, 0, h, 0, 1)
glViewport(0,0,w,h)
```

En estas líneas se configura a *OpenGL* de la siguiente manera: la proyección que se va a utilizar es `GL_PROJECTION` (primera línea), con sus medidas respectivas y sin ninguna transformación de rotación ni traslación `glLoadIdentity()`(segunda línea), la proyección es *orto-normal*(tercera línea), además se especifican las dimensiones del *"viewport"*(cuarta línea).

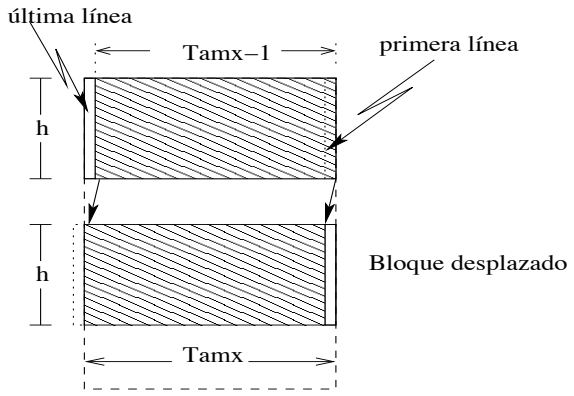
Se desea que el despliegue simule la graficación hecha por un fisiógrafo, que se idealiza como un lienzo infinito que se desplaza de derecha a izquierda, y que en su extremo derecho, hay un sistema trazador que se desplaza de arriba a bajo en forma angular, proporcional a la señal a graficar. Esto está representado en la Figura 3.9a.

Para este caso, en lugar de la punta del fisiógrafo situada en el extremo derecho, se imprimirá toda una columna vertical de tamaño h que corresponda al *periodograma* de la i-ésima ventana de tamaño R, que estará representada por la escala seleccionada (en grises o en colores).

Para hacer la simulación del fisiógrafo, se necesita una función que copie un bloque de la memoria de vídeo (correspondiente a un rectángulo de ancho h y largo $tamx - 1$) a una región (de la misma memoria de vídeo) que corresponda a un desplazamiento hacia la izquierda en una línea. como se muestra en la Figura 3.9b.



(a) De un fisiógrafo.



(b) Del espectrograma.

Figura 3.9: **Diagramas del despliegue.**

El siguiente código, en *OpenGL*, hace el efecto "trazo de fisiógrafo" para el despliegue del espectrograma correspondiente a la i-ésima ventana del instante i. El espectro se imprimirá en la primera columna de la extrema derecha del área designada por *WxWindows*, desplazándose el resto de la región gráfica hacia la izquierda.

```
glCopyPixels(x1,y1,x2,y2,GL_COLOR)
glRasterPos2i(x,y)
glDrawPixels(1,h,GL_RGBA,GL_UNSIGNED_BYTE,l.tostring())
glRasterPos2i(w-1,0)
```

Para los dos sistemas a tratar, en la clase `MyCanvasBase` la función `OnDraw()` es la encargada de todas las consideraciones arriba mencionadas para dar el efecto de fisiógrafo, ver los diagramas de las Figuras 3.10 y 3.11. Esta función es activada por el sistema gráfico, al suscitarse algún evento como: el movimiento de la ventana, la puesta de foco de la ventana, etc. También se puede invocar indirectamente al hacer una llamada a la

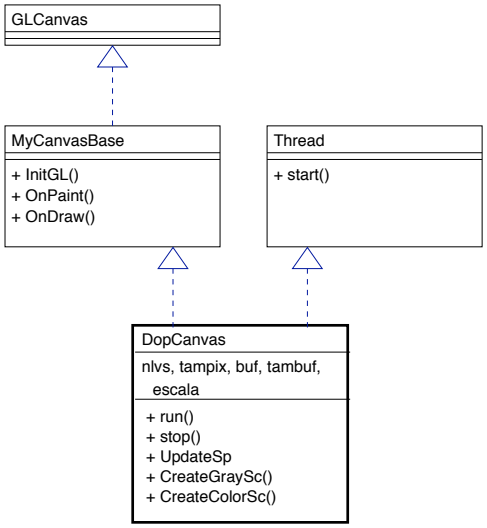


Figura 3.10: **Diagrama de la *clase activa* para el despliegue del espectro.**

función `Refresh()`. Para ambos sistemas en la función `UpdateSp()` se hace la llamada a `Refresh()`.

Ahora bien, las características esenciales para cada caso son: Para el Caso 1, se considera al despliegue asíncrono. Tal y como se destaca en el diagrama de clases de la Figura 3.10, la clase `DopCanvas` es una *clase activa*. Para el Caso 2, el despliegue es sincrónico, por tanto la clase `DopCanvas` no es activa, como se representa en el diagrama de clases de la Figura 3.11. En ambos diagramas de clases, se puede ver que la clase `MyCamvasBase`, especializa de la clase padre `GLCanvas`, todas las inicializaciones y funciones necesarias para el despliegue del espectro de la señal *Doppler* ultrasónica en forma de fisiógrafo, tal y como se señala mas arriba. Esta es la manera en que *WxWindows* incorpora en su espacio gráfico a *OpenGL*. También hay que hacer notar que en el caso de la Figura 3.10, la clase *DopCanvas* es una *clase activa* (heredera de `Thread`), que aparte de que se están manejando los eventos propios de *OpenGL*, se hace en un hilo independiente a los demás (la captura y el cálculo).

Por último, los diagramas de clases completos para cada caso los podemos ver en las Figuras 3.12 y 3.13. Se aprecia en ambos diagramas que la clase principal es `GUIFrame`. Dicha clase principal, descendiente de la clase `Frame`, en la cual residen todas las carac-

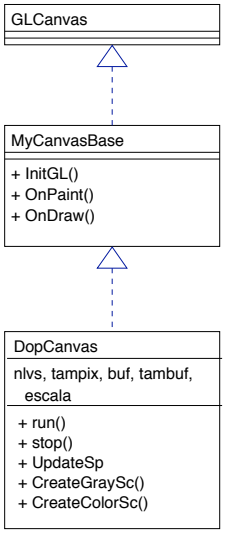


Figura 3.11: **Diagrama de la clase para el despliegue del espectro de forma sincrónica.**

terísticas propias de *WxWindows*.

3.3.5. Diagramas de objetos.

Hasta este momento hemos presentado la vista estática (diagramas de clases) de los dos casos a tratar. A continuación se presentarán los diagramas dinámicos, que presentan las actividades de las *clases activas* con los elementos de despliegue del espectro en (*OpenGL*) y los eventos de la interfaz gráfica de usuario *WxWindows*.

Caso 1

En la Figura 3.14, se ven los tres objetos activos `cap_snd` , `calc_spect` y `gldopcanvas`. `cap_snd` de forma asíncrona captura `N` bytes correspondientes a la ventana $w[n]$ (Ecuación 3.2) y los almacena en el *Queue* `buf`. `calc_spect`, lee de forma asíncrona del *Queue* `buf` los `N` bytes de la captura, hace el procesamiento respectivo para el espectro de la señal y enseguida guarda el resultado en el *Queue* `pw`. `gldopcanvas`, de forma asíncrona lee el *Queue* `pw` y extrae la línea espectral en turno, le manda al sistema gráfico la señal `Refresh()` para que este actualice el despliegue invocando la función `OnDraw()`. La sin-

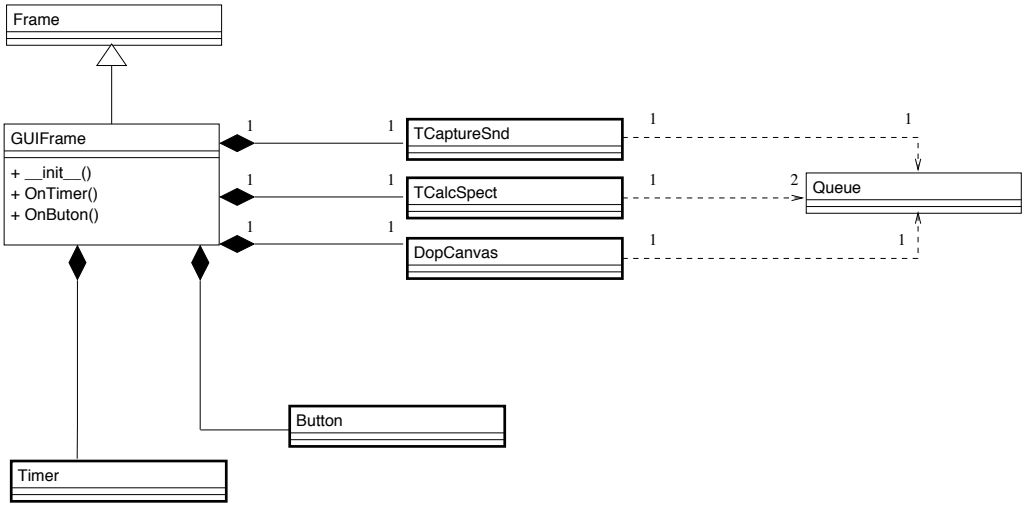


Figura 3.12: **Diagrama de clases de la aplicación con todos los eventos asíncronos.**

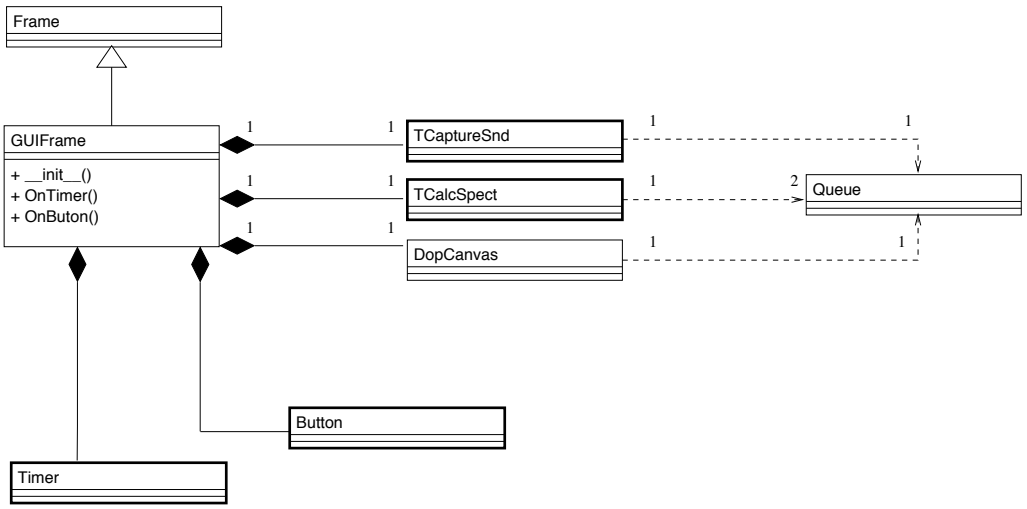


Figura 3.13: **Diagrama de clases de la aplicación con planificador.**

cronización de todos los objetos activos, lo hacen los *queues*, `buf` y `pw`, aprovechándose de que éstos son concurrentes [Martelli, 2003].

Para la visualización del tamaño de los *Queues* se hace una sincronización con `Timer`, como se ve en la Figura 3.14.

Caso 2

En el Figura 3.15, el objeto `gldopcanvas` deja de ser activo. Ahora el objeto `Timer`, invoca la función `shed()` la cual de forma asíncrona toma de `pw` la i-ésima línea espectral a ser desplegada y se la pasa al objeto `gldopcanvas` mediante el mensaje `UpdateSP(i)`. En dicha función se hace la adecuación del espectro, como se mencionó arriba, y le manda el mensaje `Refresh()` al sistema gráfico para hacer el despliegue de forma de fisiógrafo. Los objetos activos de la captura y el cálculo son los mismos que en el Caso 1.

Lo más importante del Caso 2, es la manera en que se implementó la función `shed()`. Dicha función se encarga de la sincronización del sistema, mediante un *generador* o *iterador* como también se le denomina [weightless threads, 2006]. La idea principal es que mediante la función `yield` se tenga acceso paso a paso a los valores del espectro ya calculados guardados el *Queue* `pw`. El código del generador es:

```
def generador():
    while not pw.empty():
```

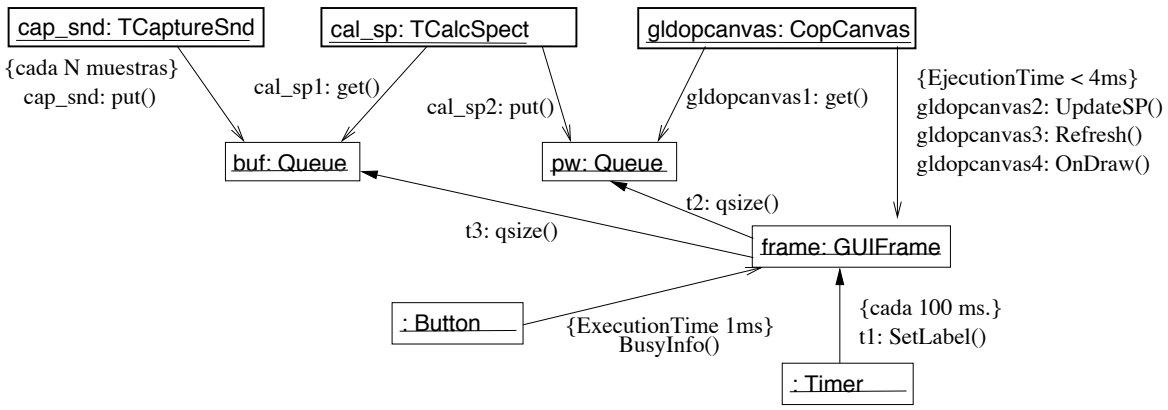


Figura 3.14: **Caso 1: Diagrama dinámico.**

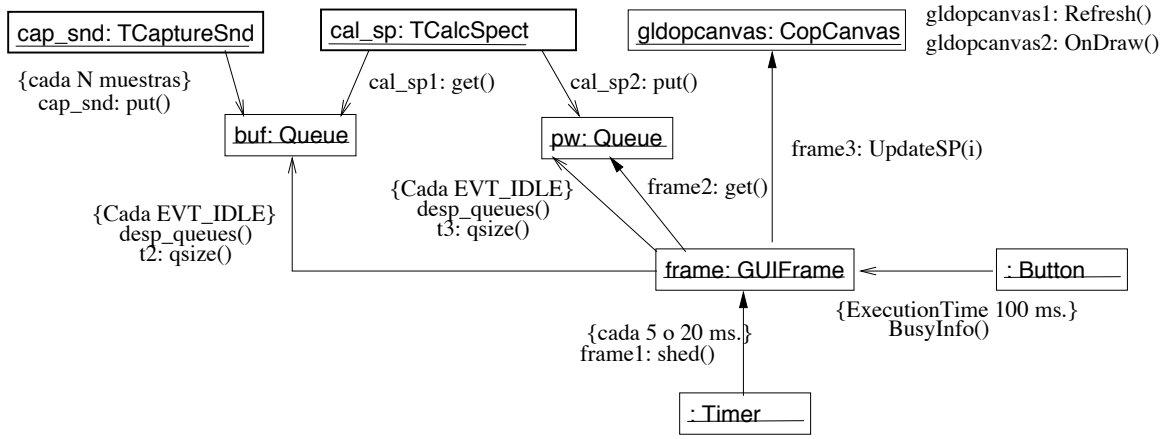


Figura 3.15: **Caso 2: Diagrama dinámico.**

```
        yield pw.get()
    yield pw.get()
```

Por tanto `shed()`, que se activa mediante el `Timer` cada determinado tiempo e irá navegando por el *Queue* `pw` como lo expresa el siguiente código

```
def shed():
    for i in generador():
        gldopcanvas.UpdateSP(i)
```

Para la visualización del tamaño de los *queues* se sincroniza con el evento `EVT_IDLE` del sistema gráfico. Esto significa que cuando el planificador de eventos del sistema gráfico esté en estado inactivo, se procederá a actualizar el estado del tamaño de `buf` y de `pw`.

3.3.6. Comportamiento del sistema con perturbaciones.

Para ambos casos, existe un botón gráfico (`Button`) como se puede ver en las Figuras 3.14 y 3.15. La acción de dicho botón es lanzar un mensaje en una ventana con una duración aproximada de 100 ms. Esta breve aparición se observa como una perturbación que repercute en el número de items almacenados en los *Queues*, los cuales, en ambos casos son constantemente monitoreados. El monitoreo en el Caso 1 es mediante un *Timer*

(ver Figura 3.14), y para el Caso 2, se hace mediante el evento *"idle"* (`EVT_IDLE`), ver Figura 3.15. Esto da una idea cuantitativa, en tiempo de ejecución, de cómo se están comportando los *Queues* concurrentes en cada sistema de "tiempo real".

3.3.7. Perfiles de ejecución de los hilos.

Para obtener una medida de los tiempos de ejecución de cada función, hilo y evento del sistema se procede a trabajar con el esquema de perfiles de ejecución (*profile*). Que *Python* tiene de forma inherente a el.

Para hacer el `profile` que incluya a los hilos, se tiene que definir la función para dicho fin. Esto se hace en el módulo `threading` mediante la función `setprofile(func_profiler)`. En donde el parámetro `func_profiler` es la función en cuestión. La definición de la función para la medición de los tiempos así como su asignación del módulo `threading` se hace en el módulo `profilethreads`. Para su utilización basta con llamar a la función `profile_on()` para activarlo y `profile_off()` para detenerlo, como se muestra en el siguiente código:

```
profile_on()
test()
profile_off()
pprint(get_profile_stats())
```

Para imprimir el resultado de las mediciones de los tiempos de cada una de las funciones utilizadas en el sistema en cuestión, se emplea la función `get_prifile_stats()` la cual es invocada dentro de la función `pprint()` como se aprecia en la última línea del código anterior.

Esto imprime los resultados en el formato que sigue:

```
('OnDraw', '<stdin>', 192): (404,
                             44.685762166976929,
```

```
0.31594800000000056,
0.11060833694911239,
0.00078204950495049639)
```

en donde se aprecian dos tuplas. La primera consta de tres elementos en donde el primer elemento es el nombre de la función en cuestión, que en este caso es `OnDraw`. El segundo elemento se destaca el archivo en el cual se encuentra la función anterior, que en este caso la denominación `<stdin>` denota que se trata del archivo principal, o el programa principal. El tercer elemento es el número de línea del archivo anterior, en donde se encuentra la función en cuestión en el archivo antes mencionado.

La segunda tupla tiene cinco elementos en donde el primero es el número de veces que fue invocado dicha función. El segundo, denota el tiempo total que la función en cuestión invirtió en todas sus n invocaciones. El tercero, es el tiempo que tardó entre invocaciones. El cuarto es el tiempo promedio de ejecución de la función. Y el último, es el tiempo promedio entre invocaciones.

3.4. Transportabilidad de los sistemas.

Hasta aquí se ha presentado el análisis y diseño de las aplicaciones. Una de las características buscadas es que las aplicaciones puedan ser ejecutadas en distintos sistemas operativos, para así poder evaluar el desempeño de sus hilos y sus eventos gráficos.

El escoger *WxWindows* como *API* para el desarrollo de aplicaciones gráficas, fue para satisfacer tal fin. Aunado al hecho de que se incorpora *OpenGL* de manera muy natural, por medio de `GLCanvas` definido en dicha *API*.

En la Figuras 3.16 y 3.17 se aprecian la aplicaciónes ejecutándose en *Mac Os X* y en *Linux* respectivamente. Se muestra el espectro de una señal que es leída por el dispositivo de audio. La señal *Doppler* capturada es de una arteria radial de un individuo masculino de aproximadamente 50 años y sano. Se muestran 13 latidos los cuales fueron captados al momento de hacer la toma de las Figuras. También se aprecian los estados de los dos

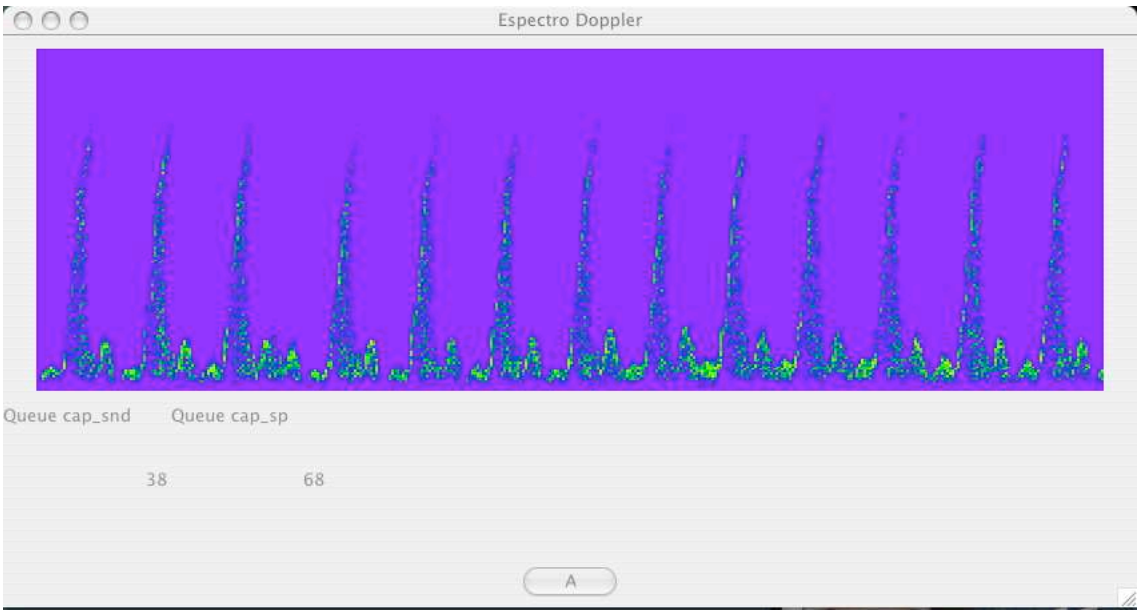


Figura 3.16: **Aplicación en *Mac Os X*.**

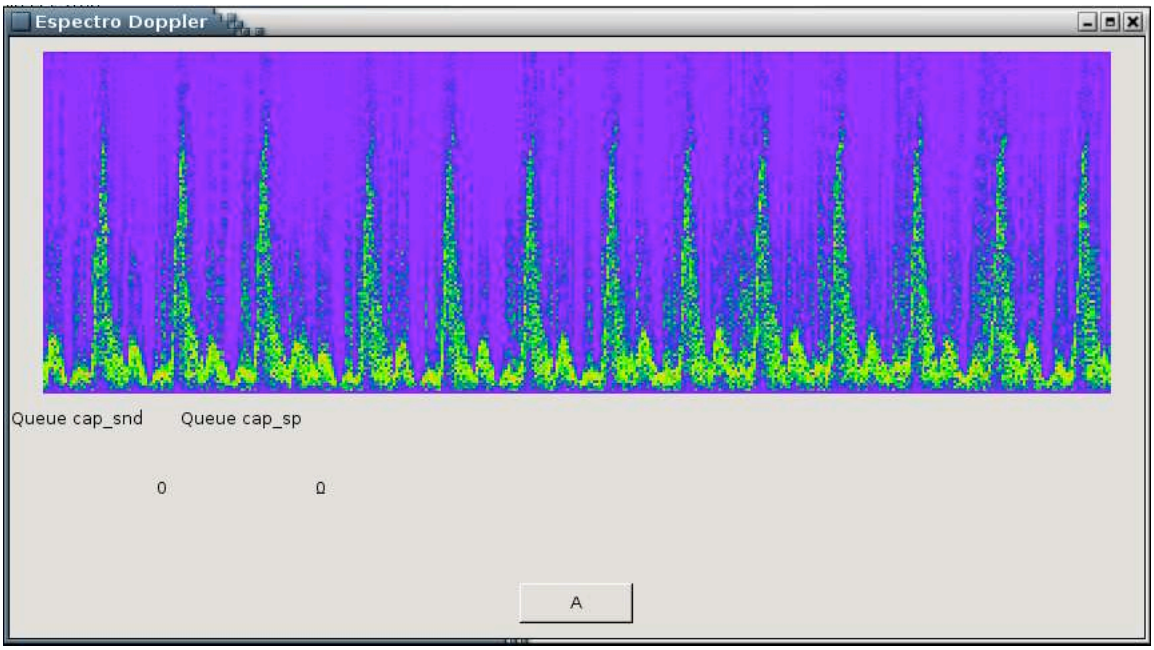


Figura 3.17: **Aplicación en *Linux*.**

Queues concurrentes y el botón para crear los artefactos gráficos.

En el próximo capítulo veremos los resultados prácticos, al correr las dos aplicaciones en dos sistemas operativos, *Linux* y *Mac Os X*.

Capítulo 4

Resultados.

En esta sección, se presentan los resultados obtenidos por el Caso 1 y el Caso 2. Las pruebas se llevaron a cabo en dos diferentes computadoras con sistemas operativos diferentes. Los Sistemas Operativos en cuestión son: *Linux* y *Mac Os X*.

Las características técnicas de cada computadora son:

Mac PowerBook G4 12", procesador PowerPC G4 a 1.33 GHz con 1.25 GB de memoria RAM, *Mac Os X* 10.3.9 (Panther), Controlador Gráfico GeForce FX Go5200 y Controlador de audio *Mac*.

Linux Procesador *Pentium IV* a 3 GHz con 1 GB de memoria RAM, versión del kernel 2.6.8 (Debian 4.0), Controlador Gráfico Intel 82915G/GV/910GL Express Chipset Family y Controlador de audio Intel 82801FB/FBM/FR/FW/FRW (ICH6 Family) High Definition Audio.

Como se mencionó en el capítulo 3, debido a las características de la señal *Doppler* ultrasónica, para visualizar el espectro de dicha señal el tiempo de la ventana debe estar en el rango de 10 a 20 ms.

Como se aprecia en la Tabla 4.1, existen algunas combinaciones para obtener los valores de de t_w deseados.

Se tomaron para los datos que se tabulan en todas las Tablas de esta sección, una

Tabla 4.1: Valores de t_w para distintas relaciones de f_m y N

N	$f_m[Hz.]$	$t_w = \frac{N}{f_m}[s.]$
512	22050	0.02322
256	22050	0.01161
256	11025	0.02322
128	11025	0.01161

frecuencia de muestreo $f_m = 22050$, ventanas de captura de tamaños $N = 256$ y $N = 512$ y un tamaño de ventana para la transformada de *Fourier* $R = 512$.

Tanto para el Caso 1 así como el Caso 2, se hicieron dos pruebas. La primera consiste en dejar que la aplicación corra libremente. En la segunda, se hacen llamadas reiterativas a un procedimiento que invoca una ventana de aviso, un "Hola Mundo", que dura unos instantes. Esto es con el fin de ver cómo se comportan los *Queues* Concurrentes cuando hay una demanda gráfica "controlada". Para todos estos casos se considera que el tamaño de los *Queues* `buf` y `pw` (Figuras 3.2 y 3.3) es muy grande, para descartar las situaciones que llamaremos de saturación.

Para el Caso 1 y el Caso 2, en las Tablas reportadas, se tabulan el número de veces que fueron ejecutados los procesos así como el tiempo promedio de ejecución del proceso en especifico.

Con las anotaciones anteriores, en las siguientes secciones se procede a explicar el Caso 1 y el Caso 2 en los diferentes Sistemas Operativos.

4.1. Comportamiento de los sistemas sin perturbaciones gráficas.

4.1.1. Caso 1

En las Tablas 4.2 y 4.3 se aprecia que los tiempos de ejecución para los procesos `cal_spectro`, `cap_snd` y `UpdateSp` son los esperados para los diferentes valores de N,

esto es para $N = 256$ es de aproximadamente de 12 ms, y para el caso de $N = 512$ es de aproximadamente 23 ms. Tiempos que están en el margen deseado, mencionado previamente. El número de ejecuciones para el caso de $N = 256$, para cada Sistema Operativo, es muy parecido. Variando solo 3 unidades para el proceso `UPdateSP` en el Sistema Operativo *Mac*. Observando el proceso `OnDraw`, los tiempos de ejecución de éste para todos los casos, no son significativos, sin embargo la relación entre su número de ejecuciones con el número de ejecuciones de los otros procesos (`cal_spectro`, `cap_snd` y `UpdateSp`) es importante verla con mayor detenimiento. Para *Mac* la relación es de 0.24 con $N = 256$, y para $N = 512$ dicha relación es de 0.49. Para *Linux* con $N = 256$ dicha relación es de 0.39, y para $N = 512$ la relación es de 0.68. Cuantitativamente hablando, para ambos Sistemas Operativos con $N = 256$, se aprecia que la ocupación del *Queue* Concurrente `pw`, va incrementando en forma indefinida mientras la aplicación esté activa. Para $N = 512$, en ambos Sistemas Operativos, el sistema no tiene el comportamiento anterior, pero hay pérdida de información, por que la relación del número de ejecuciones es menor a uno.

Tabla 4.2: R = 512, Caso 1 *Mac*

	N=256		N = 512	
	No. de ejecuciones	tiempo [ms]	No. de ejecuciones	tiempo [ms]
cap_snd	1864	12.11	1795	23.75
cal_spectro	1864	12.11	1795	23.74
UpdateSp	1861	12.10	1795	23.72
OnDraw	446	1.76	888	1.23

Tabla 4.3: R = 512, Caso 1, *Linux*

	N=256		N = 512	
	No. de ejecuciones	tiempo [ms]	No. de ejecuciones	tiempo [ms]
cap_snd	2056	11.58	1226	23.19
cal_spectro	2056	11.59	1226	23.20
UpdateSp	2056	11.59	1226	23.19
OnDraw	794	3.17	832	2.65

El comportamiento general del sistema se observa en la Figura 4.1. Las longitudes en el tiempo son proporcionales a los tiempos tabulados anteriormente para los procesos `cap_snd`, `cal_spectro` y `UpdateSp`. Las longitudes para el proceso `OnDraw` y los accesos a los *Queues* `buf` y `pw` son significativos, destacándose que son pequeños. Los rectángulos asurados de los accesos a los *Queues* significan que fue puesto un item en el *Queue* correspondiente. Como se observa el proceso `UpDateSp` hace peticiones al sistema gráfico mediante el proceso `OnDraw` y algunas de estas peticiones no son atendidas. perdiéndose el dato leído del *Queue* `pw`.

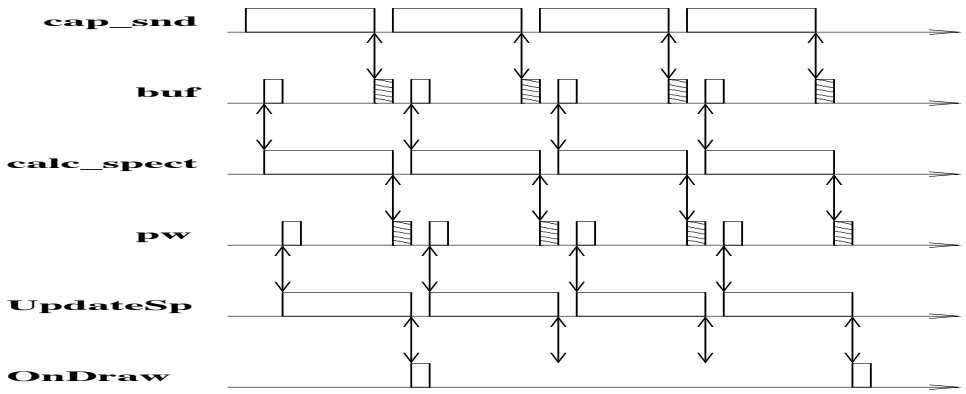


Figura 4.1: **Diagrama de tiempos para el Caso 1 sin perturbaciones gráficas.**

4.1.2. Caso 2

Para el Caso 2 se consideran dos casos más, los cuales tienen que ver con el periodo del `event Timer`, que invoca al proceso `generador`, y corresponden a los valores de 5 y 20 ms.

Para el caso en el que el `event Timer` tiene un valor de 5 ms, los valores para el Sistema Operativo *Mac* están en la Tabla 4.4 y los del Sistema Operativo *Linux* en la Tabla 4.5. Como se puede observar en estas Tablas, los valores de los tiempos no son significativos, no así el número de ejecuciones de los procesos, como en el Caso 1. Para el caso del Sistema Operativo *Mac*, y el valor de $N = 256$ se aprecia que en número de ejecuciones de `cap_snd` y de `calc_spectro` son iguales, el del `generador` es sensiblemente menor (hay

una diferencia de 278), pero en número de ejecuciones de OnDraw es de 537, dando una relación entre `cap_snd` o de `calc_spectro` de 0.37. Cuantitativamente el comportamiento es similar al del Caso 1 para el Sistemas Operativo *Mac*, el *Queue* `pw` va creciendo. Para el caso de *Linux*, lo que se destaca es que el número de ejecuciones del `generador` es superior al número de ejecuciones de los otros procesos (`cap_snd` y `calc_spectro`), en una relación al 2.7, lo cual indica un sobre muestreo de la ejecución del planificador. En cambio el número de ejecuciones del proceso `OnDraw` es sensiblemente menor al de los procesos `cap_snd` y `calc_spectro`, por 351 unidades. Esto indica que el sistema trabaja pero hay perdida de información. Para el caso en el que $N = 512$, en el Sistema Operativo *Mac* se observa un sobre muestreo del proceso `generador` con una relación del 2.67, y para *Linux* esta relación es de 5.37, pero el número de ejecuciones de `OnDraw` para *Mac* es muy parecido al de los procesos `cap_snd` y de `calc_spectro`, habiendo una diferencia solo de 3. Sin embargo para el Sistema Operativo *Linux* el número de ejecuciones de `OnDraw` es sensiblemente menor 345. Por tanto hay pérdida de información.

Tabla 4.4: R = 512, Caso 2 *Mac*, Timer = 5

	N=256		N = 512	
	No. de ejecuciones	tiempo [ms]	No. de ejecuciones	tiempo [ms]
cap_snd	1465	12.25	1147	24.01
cal_spectro	1464	12.24	1147	24.01
generador	1187	0.56	3053	0.46
OnDraw	537	2.55	1144	0.15

Tabla 4.5: R = 512, Caso 2, *Linux*, Timer = 5

	N=256		N = 512	
	No. de ejecuciones	tiempo [ms]	No. de ejecuciones	tiempo [ms]
cap_snd	1408	11.59	1433	23.21
cal_spectro	1408	11.59	1433	23.22
generador	3806	0.29	7697	0.19
OnDraw	1057	2.64	1088	2.63

Para el caso en el que el *event Timer* tiene un valor de 20 ms. los valores para el

Sistema Operativo *Mac* están en la Tabla 4.6 y los del Sistema Operativo *Linux* en la Tabla 4.7. Para el caso en el que $N = 256$ con el Sistema Operativo *Mac*, se puede ver que tanto la relación entre el número de ejecuciones del proceso `generador` con el proceso `cap_snd` o el proceso `calc_spectro` es menor a uno, y que el número de ejecuciones del proceso `OnDraw` es menor a todos, cuantitativamente el tamaño del *Queue* Concurrente va incrementándose mientras la aplicación esté activa. El mismo análisis anterior se hace para el caso del Sistema Operativo *Linux*. Para cuando $N = 512$ la relación entre el proceso `generador` y el proceso `cap_snd` o el proceso `calc_spectro` para ambos Sistemas Operativos es mayor a 2. Observado la diferencia del número de ejecuciones del proceso *OnDraw* y los otros dos procesos (`cap_snd` y `calc_spectro`), es muy pequeña, para los Sistemas Operativos estudiados (4 para *Mac* y 2 para *Linux*). Por tanto, en ningún caso hay pérdida de información.

Tabla 4.6: R = 512, Caso 2 *Mac*, Timer = 20

	N=256		N = 512	
	No. de ejecuciones	tiempo [ms]	No. de ejecuciones	tiempo [ms]
cap_snd	1497	12.24	1167	24.04
cal_spectro	1497	12.23	1167	24.02
generador	991	0.42	2391	0.53
OnDraw	480	1.98	1163	1.75

Tabla 4.7: R = 512, Caso 2, *Linux*, Timer = 20

	N=256		N = 512	
	No. de ejecuciones	tiempo [ms]	No. de ejecuciones	tiempo [ms]
cap_snd	2021	11.6	1327	23.2
cal_spectro	2021	11.6	1327	23.2
generador	2300	0.28	2890	0.22
OnDraw	1149	2.88	1325	2.64

Se destaca entonces que para el caso de *Linux* es muy susceptible a los valores seleccionados para el *event Timer*.

En la Figura 4.2 se aprecia el comportamiento general del sistema. Las longitudes en el tiempo son proporcionales a los tiempos tabulados anteriormente para los procesos `cap_snd` y `cal_spectro`. Las longitudes para el proceso `generador` y `OnDraw` y los accesos a los *Queues* `buf` y `pw` son significativos, destacándose que son pequeños. Los rectángulos asurados de los accesos a los *Queues* significan que fue puesto un item en el *Queue* correspondiente. En dicha Figura se aprecia el sobre muestreo del proceso `generador` hacia el *Queue* `pw`, no afectándose en nada la funcionamiento del sistema por que el *Queue* `pw`, al estar vacío y tener una petición de `get()`, no bloquea el proceso gestor (`generador`).

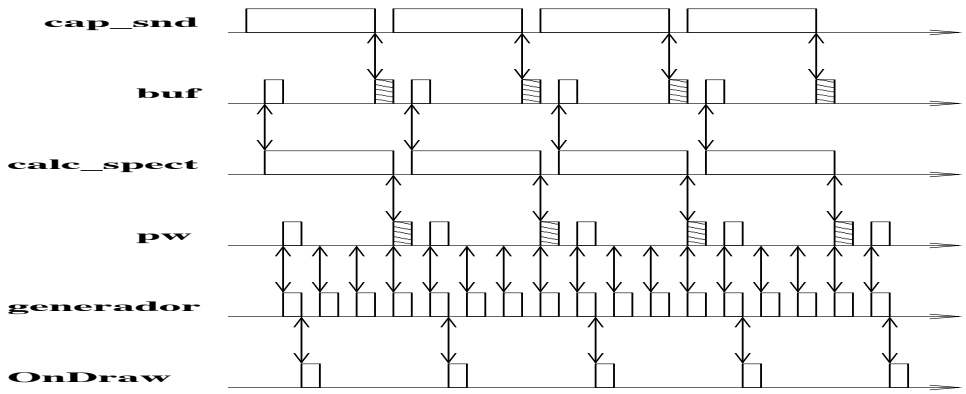


Figura 4.2: **Diagrama de tiempos para le Caso 2 sin perturbaciones gráficas.**

4.2. Comportamiento de los sistemas con perturbaciones gráficas.

El experimento consiste en hacer llamadas repetitivas a un proceso gráfico que lance una ventana y muestre un mensaje. Este proceso tiene una duración finita y pequeña, de unos mili-segundos. Este proceso tiene el mismo comportamiento al del botón de la aplicación, el cual se puede ver esquemáticamente en el diseño en la sección 3.15.

Al hacer experimentaciones con el botón de la aplicación para el Caso 1, se aprecia notoriamente pérdidas de información al no funcionar el almacenamiento del *Queue* `pw`.

Este comportamiento se esquematiza en la Figura 4.3. En dicha Figura se aprecia que hay una serie de peticiones hechas por `UpdateSp` al *Queue* `pw` que son exitosas, pero las llamadas al sistema gráfico mediante la función `OnDraw`, por el proceso `UpDateSp` son infructuosas.

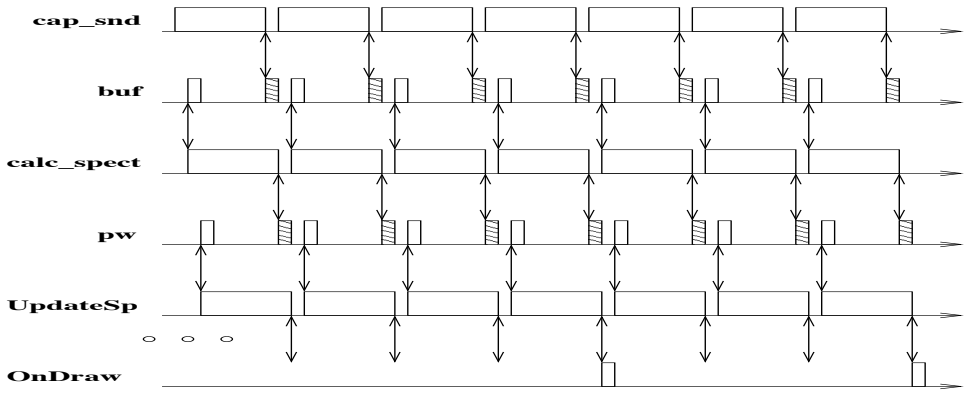


Figura 4.3: **Diagrama de tiempos para el Caso 1 con perturbaciones gráficas.**

Para el Caso 2 se aprecia un buen funcionamiento del sistema, observándose un aumento de items en el *Queue* `pw` y su posterior despacho del mismo *Queue* por el proceso gráfico hasta llegar al equilibrio en un lapso razonable de tiempo.

Para el Caso 2 se hará el mismo procedimiento antes señalado, con un valor de `event Timer` de 5 ms, un tamaño de ventana $N = 512$ y el tamaño $R = 512$ para la transformada de *Fourier*.

Para el Sistema Operativo *Mac*, el comportamiento del *Queue* `pw` en el tiempo se aprecia en la Figura 4.4. Se distinguen subidas súbitas correspondientes a el lanzamiento de la perturbación gráfica arriba descrita. El tiempo que tarda en cargarse el *Queue* pw con 63 items es de 3 ms. Estos 63 items tardan en descargarse 532 ms. correspondiente a una pendiente de descarga de $m = -0{,}12$. Los eventos gráficos se repiten cada 800 ms.

Para el Sistema Operativo *Linux*, el comportamiento del *Queue* `pw` se puede ver en la figure 4.5. Los eventos gráficos se repiten cada 800 ms. El *Queue* `pw` se carga de 44 items en 2 ms. Estos 44 items se descargan en 312 ms. correspondiente a una pendiente de descarga $m = -0{,}14$.

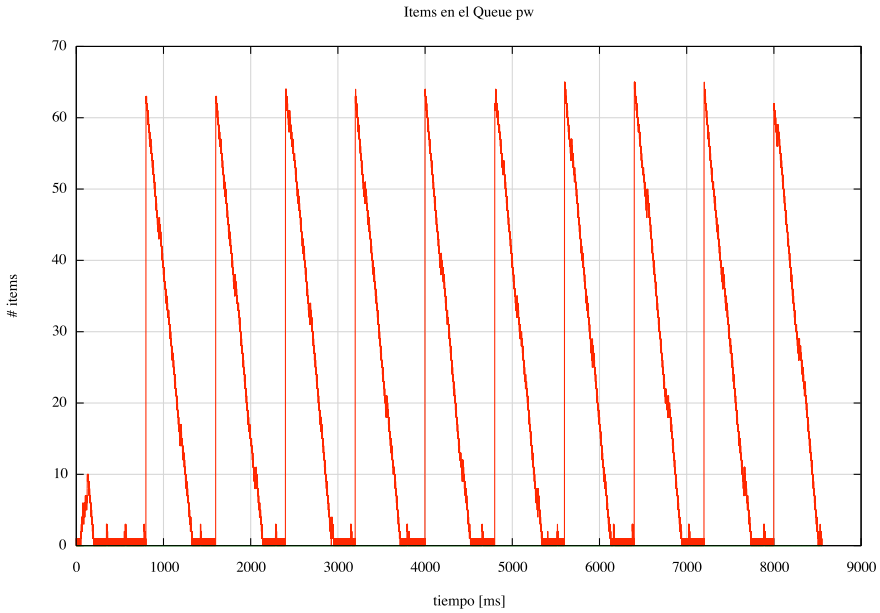


Figura 4.4: ***Mac*: Actividad de los *Queues* en el tiempo.**

En la Figura 4.6 se puede ver en detalle uno de los dientes de sierra que se presentan en las Figuras 4.4 y 4.5. El análisis que a continuación se da, en particular de un diente de sierra, se aplica a lo general para los Sistemas Operativos en cuestión (*Mac Os X* y *Linux*, Figuras 4.4 y 4.5). Si se define el número de items que ingresan al *Queue* por unidad de tiempo como λ y el número de items que se atienden por unidad de tiempo como μ [Gross and Harris, 1985], de las Figuras antes señaladas se observa que las pendientes de los dientes de sierra son negativas y constantes para cada caso, como además se discutió anteriormente, entonces se asegura que $\lambda < \mu$. Esto implica que el consumo es mayor a la inserción de elementos en el *Queue*. Se distinguen ciertas regiones de la gráfica de la Figura 4.6 con un asurado en gris, en las cuales se observa que ahí la relación es: $\lambda > \mu$. En éstas regiones, en un lapso de unos cuantos mili-segundos hay mas ingresos que consumos y esto se debe a que el Sistema Operativo en el que se está trabajando no está dedicado cien por ciento a la aplicación en cuestión. Se observan también las regiones de asurado rojo en la misma Figura 4.6, de unos cuantos mili-segundos en los cuales la relación es de igualdad ($\lambda = \mu$), y por tanto número de items en el *Queue* no cambia en ese lapso de

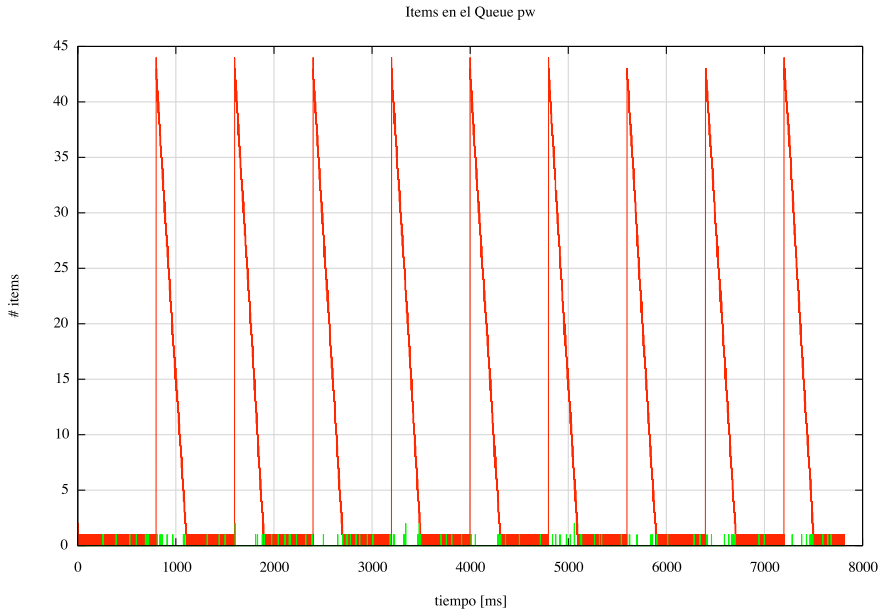


Figura 4.5: ***Linux*: Actividad de los *Queues* en el tiempo.**

tiempo. Lo importante es que el comportamiento global del *Queue* es determinístico y se cumple en general la relación $\lambda < \mu$. Además de que al presentarse alguna perturbación gráfica, el comportamiento del *Queue* Concurrente sigue siendo determinístico y el análisis es como el de cualquier *Queue* con la relación $\lambda < \mu$ que tiene un determinado número de items como condición inicial.

Por úntimo, para completar el par de diagramas de tiempos para el Caso 2, sin perturbaciones gráficas Figura 4.2, se presenta el diagrama de tiempo con perturbaciones gráficas, Figura 4.7. En dicha Figura se esquematiza el hecho de que al estar presente la perturbación gráfica, solo hay inserciones en el *Queue* `pw`, y al término de dicha perturbación el *Queue* empieza a trabajar con las condiciones $\lambda < \mu$ y un número de items en él como condición inicial, que para el cado de *Linux* es 44 items (ver Figura 4.5) y para el caso de *Mac Os X*es de 63 (ver Figura 4.4).

En el presente capítulo se presentaron dos aproximaciones, el Caso 1 y el Caso 2, las cuales se estudiaron con y sin perturbaciones gráficas para los dos diferentes Sistemas Operativos en estudio: *Linux* y *Mac Os X*. El Caso 1 se modeló con todos los procesos

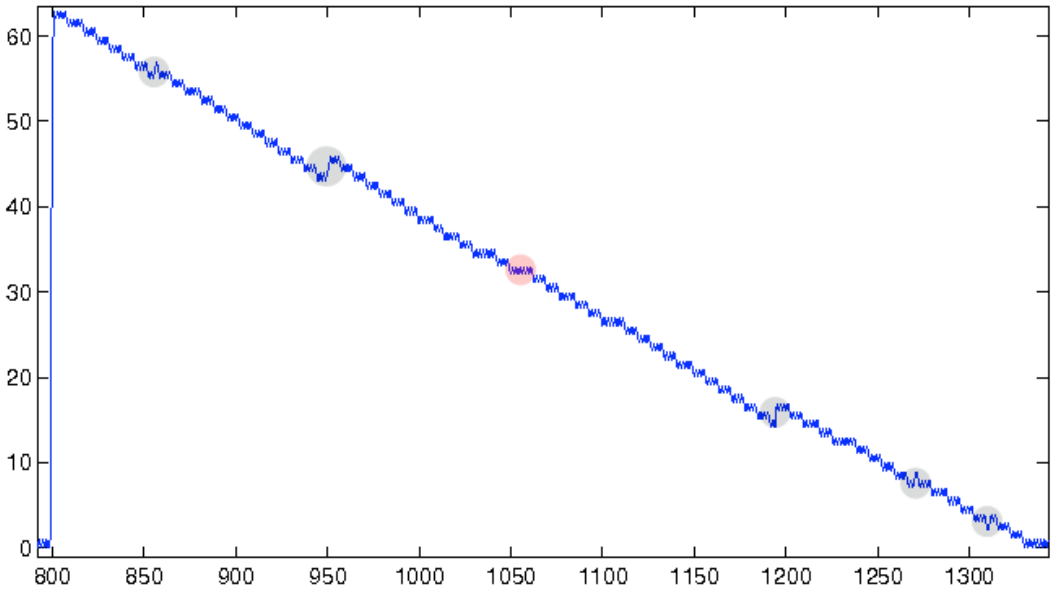


Figura 4.6: **Detalle del comportamiento del *Queue* `pw` en el tiempo.**

involucrados para la captura procesamiento y la graficación (`cal_spectro`, `cap_snd` y `OnDraw`) como hilos. El Caso 2 se modeló con `cal_spectro` y `cap_snd` como hilos y para la parte gráfica se implementó un planificador mediante un *generador*. Con base a los resultados obtenidos se muestra que para el caso Caso 2 los comportamientos de los *Queues* concurrentes son deterministas, aún cuando existan perturbaciones graves en el sistema. Por tanto, debido al sobre muestreo que se tiene, el comportamiento general del

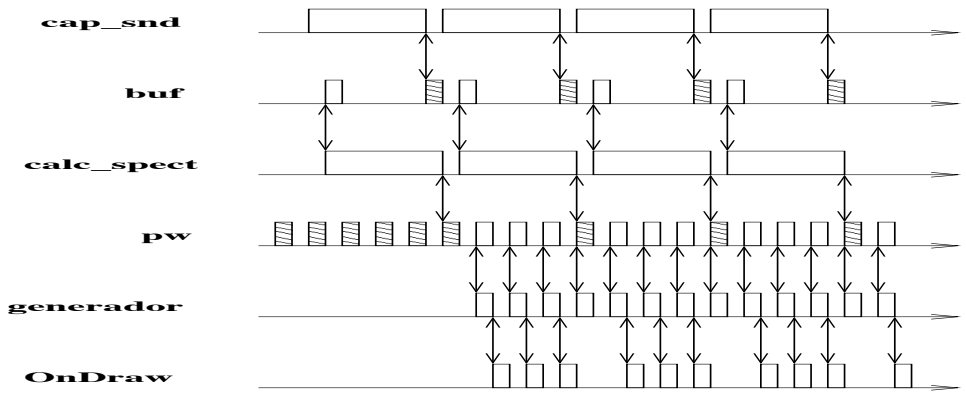


Figura 4.7: **Diagrama de tiempos para el Caso 2 con perturbaciones gráficas.**

sistema es de *Tiempo Real Suave*.

Capítulo 5

Conclusiones.

5.1. Conclusiones generales.

En el este trabajo se presentaron dos diferentes técnicas para sincronizar hilos y eventos gráficos mediante un lenguaje interpretado, el cual tiene las siguientes características: Manejar intrínsecamente los conceptos de concurrencia mediante hilos, contar con estructuras de datos avanzadas que manejen concurrencia y manejar el concepto de *generador* además de incorporar componentes de software, como *frameworks* para la creación de interfaces gráficas de usuario.

El Caso 1 consiste en la implementación del sistema haciendo que la clase encargada del despliegue sea una *clase activa* mediante herencia múltiple, entre la `clase Thread` y la `clase GLCanvas`. El Caso 2 consiste en la creación de un planificador implementándose en la clase principal de la aplicación gráfica un *generador*.

La utilización de *Queues* concurrentes aunado a los *generadores* (Caso 2) es la mejor metodología para obtener un sistema concurrente y transportable para la adquisición, procesamiento y despliegue de señales en Tiempo Real Suave. Cumpliéndose así el objetivo principal de esta tesis.

La conjetura para el Caso 1, es que el sistema gráfico (vía el *WxWindows*) sincronice los eventos gráficos al ser demandados mediante la función `OnDraw` (o mediante `Refresh`).

Esto no sucede porque el proceso principal encargado de manejar los eventos gráficos es sincrónico. Esto trae como consecuencia pérdidas de cantidades de elementos de procesamiento (muestras de datos) que tienen un efecto visible en la respuesta gráfica del caso de estudio.

En el Caso 2, el *generador* permite controlar los eventos en el ciclo principal de la interfaz gráfica de usuario, obteniéndose de este modo la sincronización entre los eventos gráficos y los hilos: *captura* y *proceso*. Este control se lleva a cabo mediante el objeto que despliega, y tiene las características de un planificador de los eventos gráficos de la aplicación. En especifico el *event Timer* (Figura 3.14) es quien da la sincronía al sistema para que el despliegue se ejecute, y que la aplicación obtenga un Tiempo Real Suave sin la necesidad de la sincronía de un Sistema Operativo en Tiempo Real. El `EVT_IDLE` (Figura 3.14) se aprovecha para el monitoreo de los *Queues* concurrentes en la interfaz gráfica, aún cuando no tenga un efecto directo en el desempeño del sistema.

Como se puede ver en las Tablas 4.4,4.5,4.6 y 4.7, debido a que el proceso `generador` presenta un sobre-muestro, la aplicación es capaz de recuperarse cuando un evento gráfico externo es atendido por el sistema gráfico (siempre y cuando no se exceda el tamaño del *Queue*), lo que genera una supuesta pérdida de muestras que en realidad están almacenadas en los *Queues* concurrentes y que no presentan mayor efecto dada la característica del `generador`, por lo que se simplifica en un problema de *muestreo*. Dicha recuperación parte del intercambio del *Queue* concurrente y el control que ejerce sobre el ciclo principal del sistema de atención de eventos gráficos. Estos resultados se presentaron en [Cańdenas-Flores et al., 2007].

5.2. Trabajo Futuro.

Hay que considerar que el análisis y diseño hecho en el presente trabajo de tesis, es orientado a objetos y no es la única manera de modelar un sistema de procesamiento digital de señales. Por ejemplo se puede hacer un modelado mediante una programación visual

como el que se hace en *Simulink* de *Matlab*, que esquemáticamente sería muy parecida a la presentada en la Figura 1.1. El tipo de análisis que resulta de la programación visual, se le conoce como *redes de procesos de flujo de datos* (Dataflow Process Networks) [Lee and Parks, 1995]. Éste es un *modelo de computación* en donde un cierto número de procesos concurrentes se comunican a través de canales *FIFO* (Queues), en dónde además la escritura hacia los canales no bloquea a los demás procesos involulucrados, pero la lectura si los bloquea. Al modelo anterior también se le conoce como Redes de Procesos de Kahn [Kahn, 1974]. Al representarse los bloques de la Figura 1.1 como procesos que se comunican mediante las flechas, que representan los *FIFOs*, tenemos una equivalencia con el modelado obtenido mediante un análisis y diseño orientado a objetos. Si aunado a lo anterior se involucra, un marcador de tiempo como el `evtTimer` utilizado en esta tesis, estamos hablando del modelo de computación conocido como *eventos discretos*. Dicho modelo lleva a un análisis semántico, por medio del cual [Lee, 2006a] propone como unidad de diseño robusta para el manejo de la concurrencia al los *actores*.

Es de llamar la atención que se esté ejecutando la aplicación en cuestión en un sistema operativo común, y se llegue a tener un Tiempo Real Suave. El principal trabajo de los sistemas operativos de Tiempo Real es poner en serie y de manera determinista a los procesos que estén activos en él. Por el resultado obtenido en el presente trabajo de tesis, se pone de manifiesto que hay cierta "holgura" para el manejo de las tareas utilizando los *Queues* concurrentes y el esquema de *generador*, en cuanto a la serialización y el determinismo de la aplicación. En [Gravangne et al., 2005] se propone un método de análisis para estudiar tal "holgura" mediante el estudio de un controlador en tiempo real utilizando la teoría de las *escalas de tiempo* [Bohner and Peterson, 2001]. Para analizar el mismo fenómeno, pueden seguir las ideas expuestas en el análisis hecho en el entorno de redes, en especial en los protocolos de TCP, en cuanto al los tamaños de las ventanas que se utilizan para la transmisión (mtu Maximum transmission unit) por medio de ecuaciones diferenciales estocásticas [Mistra et al., 2000].

Bibliografía

[Aho et al., 2007] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2007). *Compilers: Principles, Techniques, & Tools*. Addison Wesley, second edition.

[Bohner and Peterson, 2001] Bohner, M. and Peterson, A. (2001). *Dynamic Equations on Time Scales*. Birkhauser.

[Booch, 1994] Booch, G. (1994). *Object Oriented Analysis and Design*. The BenjaminCommings Publishing Company Inc.

[Booch et al., 1999] Booch, G., Rumbaugh, J., and Jacobson, I. (1999). *El Lenguage Unificado de Modelado*. Addison Wesley.

[Cańdenas-Flores et al., 2007] Cańdenas-Flores, F., Benıt́ez-Peŕez, H., and Garcıá-Nocetti, F. (2007). Study of concurrent queued system for soft real time purpose: Bioengineering case study. In *IEEE CERMA*, pages 68–73.

[Cárdenas Flores, 1996] Cárdenas Flores, F. (1996). Python: Un lenguaje oriantado a objetos alternativo. *Soluciones Avanzadas*, (39):46–50.

[Evans and McDicken, 2000] Evans, D. H. and McDicken, N. (2000). *Doppler Ultrasound; Phisics, Instrumentation and Signal Processing*. Jhon Hilley & Sons, LTD.

[Fish, 1991] Fish, P. (1991). Non-stationary broadening in pulsed doppler scpectrum meassurements. *Ultrasound in Medicine and Biology*, (17):147–155.

[Fuentes et al., 2006] Fuentes, M., Sotomayor, A., García, F., Moreno, E., and Acevedo., P. (2006). Design and construction of a blood flow detector probe for a continuous wave bidirectional doppler ultrasound system. *Ingeniería Investigación y Tecnología*, (2):97–103.

[García et al., 2006] García, F., Moreno, E., Solano, J., Barragán, M., Sotomayor, A., Fuentes, M., and Acevedo., P. (2006). Design of a continuos wave blood flow bi-directional dopper system. *Ultrasonics*, (44):e307–e312.

[Goetz et al., 2006] Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006). *Java Concurrency in Practice*. Addison Wesley Professional.

[Gravangne et al., 2005] Gravangne, I. A., Davis, J. M., and Marks, R. J. (2005). How deterministic must a real-time controller be? In *IEEE RSJ International Conference on Inteligent Robots and Systems*.

[Gross and Harris, 1985] Gross, D. and Harris, C. M. (1985). *Fundamentals of Queueing Theory*. John Wiley & Sons, second edition.

[Guyton, 1987] Guyton, A. C. (1987). *Fisiología Médica*. Interamericana, México D.F, vi edition.

[Hristu-Varsakelis et al., 2005] Hristu-Varsakelis, D., Levine, W. S., Alur, R., K.-E. Arzen, J. B., and Henzinger, T., editors (2005). *Handbook of Networked and Embedded Control Systems*. Birkhäuser.

[Jensen, 1996] Jensen, J. A. (1996). *Estimation of Blod Velocities Using Ultrasound A signal processing Approach*. Cambridge University Press.

[Kahn, 1974] Kahn, G. (1974). The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475.

[Kopetz, 1997] Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. The International Series in Engineering and Computer Science. Springer.

[Kuo et al., 2006] Kuo, S. M., Lee, B. H., and Tian, W. (2006). *Real-Time Digital Signal Processing: Implementations adn Applications*. Willey, second edition edition.

[Lee, 2006a] Lee, E. A. (2006a). Concurrent semantics without the notions of state of state transitions. In *FORMATS*.

[Lee, 2006b] Lee, E. A. (2006b). The problem with threads. *IEEE, Computer*, pages 33–42.

[Lee and Parks, 1995] Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Procedings of the IEEE*, 83(5):773–799.

[Lee and Zhao, 2007] Lee, E. A. and Zhao, Y. (2007). Reinventing computing for real time. *LNCS 4322*, pages 1–25.

[Lutz, 2001] Lutz, M. (2001). *Programming Python*. O'Reilly.

[Martelli, 2003] Martelli, A. (2003). *Python in a Nutshell*. O'Reilly.

[Mistra et al., 2000] Mistra, V., Gong, W.-B., and Towsley, D. (2000). Fluid-based analysis of a network of aqm routers supporting tcp flows with an application to red. In *SIGCOMM'00*.

[Mitra, 2001] Mitra, S. K. (2001). *Digital signal processing*. McGraw-Hill.

[Neider et al., 1993] Neider, J., Davis, T., and Woo, M. (1993). *OpenGL programming Guide*. Addison Wesley.

[Oppenheim et al., 1999] Oppenheim, A. V., Schafer, R. W., and Buck, J. R. (1999). *Discrete-time signal processing*. Printice Hall.

[Ousterhout, 1996] Ousterhout, J. (January 25, 1996). Why threads are a bad idea. Recuperado el 20 de oct 2006. Platica invitada, USENIX. http://home.pacbell.net/ouster/threads.pdf.

[Proakis and Manolakis, 1998] Proakis, J. G. and Manolakis, D. G. (1998). *Tratamiento digital de senãles. Principios, algoritmos y aplicaciones.* Prentice Hall.

[Process Wikipedia, 2006] Process Wikipedia (Recuperado el 20 oct. 2006). Process (computing). http://en.wikipedia.org/wiki/Computer_process.

[Python API, 2006] Python API (Recuperado el 20 oct. 2006). Python api reference manual. http://www.python.org.

[Robbins and Robbins, 2003] Robbins, K. and Robbins, S. (2003). *Unix Systems Programming: Communication, Concurrency and Threads.* Prentice Hall PTR, 2 edition edition.

[Stevens, 1992] Stevens, W. R. (1992). *Advanced Programming in the UNIX Environment.* Addison Wesley, Reading, Massachusetts.

[Tanenbaum, 1992] Tanenbaum, A. S. (1992). *Modern Operating Systems.* Prentice Hall.

[Tanenbaum, 2006] Tanenbaum, A. S. (2006). *Structured computer organizacion.* Pearson, Prentice Hall, 5th edition.

[Thread Wikipedia, 2006] Thread Wikipedia (Recuperado el 20 de oct. 2006). Thread (computer science). http://en.wikipedia.org/wiki/Thread_%28computing%29.

[weightless threads, 2006] weightless threads (Recuperado el 20 de oct. 2006). Charming python: Implementing "weightless threads"with python generators. http://www-128.ibm.com/developerworks/linux/library/l-pythrd.html.