ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

FACULTAD DE INGENIERÍA EN ELECTRICIDAD Y COMPUTACIÓN

“Diseño e Implementación de Procesos de Captura de Datos en Soft y Hard Real-Time usando RTAI"

INFORME DE MATERIA DE GRADUACIÓN

Previa a la obtención del Título de:

INGENIERO EN CIENCIAS COMPUTACIONALES

ESPECIALIZACIÓN SISTEMAS MULTIMEDIA

INGENIERO EN TELEMÁTICA

Presentado por

HENRY ANTONIO LOMAS ASCENCIO

DENISSE ADRIANA OCHOA CEVALLOS

GUAYAQUIL - ECUADOR

2013

AGRADECIMIENTO

Agradecemos a Dios por bendecir nuestro camino, a nuestras familias por ser nuestro soporte incondicional y a esas personas que se han vuelto parte especial de nuestras vidas.

DEDICATORIA

Dedico este trabajo a mi madre por ser el ejemplo de vida que me ha inspirado a crecer.

Henry Lomas A.

Dedico este trabajo a mi familia, en especial a mi mamá por ser el motor que impulsa mi vida.

Denisse Ochoa C.

TRIBUNAL DE SUSTENTACIÓN

Dr. Daniel Ochoa
PROFESOR DE LA MATERIA DE GRADUACIÓN

Msc. María A. Álvarez Villanueva
PROFESOR DELEGADO POR EL DECANO DE LA FACULTAD

DECLARACIÓN EXPRESA

"La responsabilidad del contenido de este Informe de Graduación, nos corresponde exclusivamente; y el patrimonio intelectual de la misma, a la Escuela Superior Politécnica del Litoral"

(Art. 12 Reglamento de Graduación de la ESPOL)

Henry Lomas Ascencio.

Denisse Ochoa Cevallos.

RESUMEN

El siguiente trabajo de tesis trata del control de un sistema simple conectado en red siendo monitoreado por un programa informático que desarrollamos con el uso de la herramienta RTAI.

Combina el uso de técnicas de programación y un sistema de control simulado. El esquema es la básica relación de cliente-servidor, siendo el servidor el sistema que simula un OPC, que nos abstrae de la conexión con un PLC por ejemplo, y el cliente es nuestro sistema informático desarrollado que permite leer los datos del OPC en tiempo real.

Para poder hacer la selección de todas nuestras herramientas para el desarrollo final hemos realizado varios experimentos que implica la comparación entre lenguajes C y Python con el uso de RTAI, así como la comparación entre los protocolos de comunicación OPC DA y XML.

Al final contamos con el sistema de control óptimo con las herramientas seleccionadas y las conclusiones del mismo.

ÍNDICE GENERAL

ANEXOS

1. **ANEXO A:** Instalación de RTAI.
2. **ANEXO B:** Código en C con herramientas de RTAI para comparar RTAI versus Linux estándar.
3. **ANEXO C:** Código en C que servirá para comparar mejoría entre usar RTAI o un Linux estándar.
4. **ANEXO D:** Código en Python con herramientas RTAI que comparar versus código en C con herramientas RTAI.
5. **ANEXO E:** Código en Python con herramientas RTAI que realiza conección a un servidor XML-DA.
6. **ANEXO F:** Código en Python con herramientas RTAI que realiza conección a un servidor OPC DA.
7. **ANEXO G:** MAKEFILE para compilar un programa con RTAI.
8. **ANEXO H:** Script RUN para ejecutar un programa con RTAI.

GLOSARIO

API: Por sus siglas en inglés que provienen de Application Programming Interface. Es una librería que incluye: estructuras de datos, clases de objetos, variables, etc.

Kernel: Es el núcleo de un sistema operativo, la parte principal. Es donde van a ser atendidas los procesos, las interrupciones, maneja la comunicación entre el hardware y el software entre algunas de sus responsabilidades.

Kernel vainilla: En inglés vanilla kernel. Es un kernel que contiene las librerías y funciones básicas implementadas por los desarrolladores de Linux y no ha sido modificado por las diferentes comunidades que desarrollan sistemas operativos. En nuestro proyecto es el que nos va a servir de microkernel para RTAI.

ÍNDICE DE FIGURAS

INTRODUCCIÓN

Los sistemas informáticos que necesitan un alto grado de precisión para sus aplicaciones en las industrias o los campos médicos dependen no solo en la efectividad y veracidad de sus datos, sino también de la velocidad con la que son presentados, es decir, el tiempo de respuesta de estos sistemas. La importancia de la información contenida en estos sistemas si no son proporcionados antes de que se cumpla el tiempo máximo de respuesta los datos no serán útiles por muy acertados que sean.

Es aquí donde yace la importancia de los Sistemas en Tiempo Real: en lo crítico que puede resultar entregar a tiempo los datos que conformen una información crítica o muy necesaria. Los Sistemas en tiempo real deben cumplir básicamente con los mismos requerimientos de un sistema estándar: que sus datos sean precisos y correctos, pero dentro del tiempo máximo de respuesta.

En este trabajo utilizaremos Técnicas de Programación en Tiempo Real que sumado a un conjunto de librerías que facilitan cumplir

con las exigencias que comprende un sistema en tiempo real podremos optimizar un sistema de control simulado que requiere hacer uso de estas ventajas en su campo de aplicación.

Para nuestro acometido utilizamos el sistema operativo Ubuntu 9.04, un kernel vainilla versión 2.6.24.7 y una versión de RTAI 3.7 para la instalación de RTAI poder hacer uso de sus librería, PyOPC para simular un servidor XML-DA y el software Matrikon para simular un servidor OPC DA.

Los capítulos están presentados de la siguiente manera:

Capítulo 1: Generalidades

Planteamos los objetivos generales, objetivos específicos, alcances y limitaciones del proyecto.

Capítulo 2: Marco Teórico

Se introduce los conceptos de qué es un sistema en tiempo real, sus requerimientos, sus especificaciones además de qué es RTAI junto a su API y de cómo realizar un programa usando sus librerías.

Capítulo 3: Herramientas

Exponemos las herramientas que fueron utilizadas en el proyecto, sus diferencias, sus similitudes y sus alcances.

Capítulo 4: Resultados

Presentamos los gráficos obtenidos a lo largo de los experimentos realizados con su respectiva conclusión.

CAPITULO 1

1. GENERALIDADES

1.1. Objetivos

1.1.1. Objetivo general

- Crear una aplicación que simule un sistema de comunicación de control aprovechando las ventajas de las técnicas de programación en tiempo real y de las librerías dedicadas a este uso.

1.1.2. Objetivos específicos

- Estudiar los Sistemas OPC y sus protocolos de comunicación para poder elegir el más apropiado para nuestro objetivo.
- Crear un sistema simulado OPC con el que pueda comunicar satisfactoriamente un cliente OPC y un servidor OPC.

Estos objetivos se realizarán a lo largo de los experimentos que se han definido para crear la aplicación final.

1.2. Alcances y Limitaciones del Proyecto

- El sistema OPC es simulado en software.
- Se simulan dispositivos conectados al servidor y permite al cliente realizar peticiones.

CAPÍTULO 2

2. MARCO TEÓRICO

2.1. Sistemas Operativos

Un sistema operativo es un proceso que actúa de intermediario entre el usuario y el hardware del computador. Se encarga de la administración de los dispositivos, administración de los procesadores, ejecución de programas, servicio de archivo, manejo del uso de memoria, entre algunas de sus tareas.

Sus componentes son: núcleo o kernel, intérprete de comandos, llamadas al sistema, aplicaciones de usuario y HAL (Hadware Abstraction Layer). [1]

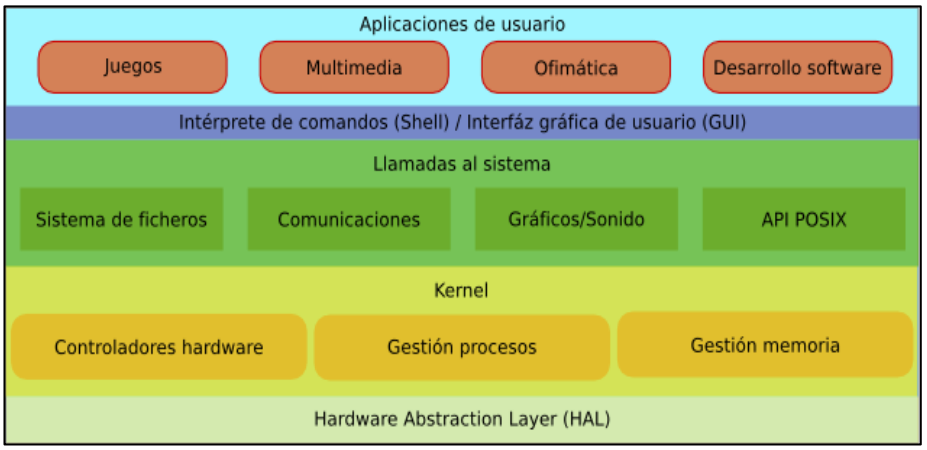


Figura 2.1 Componentes de un sistema operativo [1]

En la actualidad podemos encontrar varios sistemas operativos, por ejemplo: Windows, MacOS, Linux, entre otros.

Un tipo especial de sistemas operativos es de tiempo real que presentamos en la Sección 2.5.

2.2. Kernel

El kernel o núcleo es la parte fundamental de un sistema operativo, se encarga de la comunicación entre las aplicaciones y el hardware.

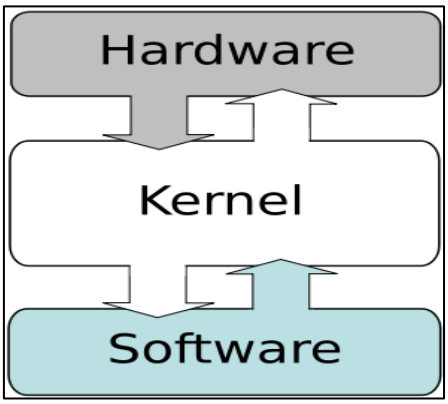


Figura 2.2 Kernel [2]

El kernel decide la ejecución de los programas y administra el uso del hardware como: la memoria del sistema, los periféricos de entrada y salida, dispositivos de almacenamiento entre otros.

Existen varios tipos de kernel: monolíticos, microkernels, exokernels, y los híbridos.

El microkernel es de nuestro principal interés, lo utilizamos para el desarrollo de nuestra tesis. El microkernel se abstrae un poco del hardware, provee un conjunto de llamadas mínimas al sistema para implementar servicios básicos como espacios de direcciones, comunicación entre procesos y planificación básica. [3]

El kernel tiene dos tipos de trabajo: modo usuario o modo kernel.

El modo usuario no tiene acceso directo a las estructuras de datos del kernel o al hardware directamente, sino mediante el uso de las APIs del sistema para tener acceso.

El modo kernel, por el contrario, tiene potestad de ejecutar todas las instrucciones del CPU, hacer referencia a cualquier dirección de memoria y tiene absoluto acceso directo con el hardware.

Un proceso por lo general corre en modo usuario pero puede cambiar a modo kernel si requiere algún servicio del kernel, por ejemplo, si el proceso necesita invocar una llamada al sistema.

2.2.1. Planificador

El planificador es el encargado de distribuir el tiempo de ejecución de los procesos en el CPU. Esto es importante porque en algún instante varios procesos

pueden estar listos para ser ejecutados pero sólo uno puede acceder al uso del CPU.

Cuando un proceso se ha ganado el uso del CPU este puede ser “expulsado” por el planificador si su tiempo de ejecución vence o si este proceso necesita de la ejecución de otro. Para controlar cuando un proceso está por vencer el planificador se vale del uso de un temporizador que se ejecuta en cada intervalo de tiempo.

La selección de qué proceso se va a ejecutar depende de la política de planificación pre-establecida.

Las políticas de planificación pueden ser: FIFO (First In, First Out), Round Robin, planificación por prioridades, entre otras. Lo habitual es usar políticas mixtas, es decir, podemos estar frente al cumplimiento de más de una política a la vez.

Los algoritmos de planificación pueden ser: expropiativos o apropiativos y no expropiativos o no apropiativos. Los expropiativos asignan un tiempo de

ejecución a cada proceso hasta que acabe y se ejecute el siguiente, permite expulsar a un proceso en ejecución si llega otro de mayor prioridad que necesita ejecutarse. Los no expropiativos permiten ejecutan el proceso hasta que acabe su trabajo, es decir, una vez que les llega el turno de ejecutarse no dejarán libre la CPU hasta que terminen o se bloqueen. [4]

2.3. Sistemas en tiempo real

El contenido de esta Sección y hasta la Sección 2.4.2.1 está basado en el libro “Sistemas de tiempo real” [5]

Un sistema en tiempo real es un sistema informático donde no sólo se requiere que el resultado sea correcto sino que debe ser generado en un específico periodo de tiempo límite, conocido en la jerga informática como *deadline*. Los resultados después de cumplirse dicho límite de tiempo pueden ser correctos pero ya no son válidos porque el proceso no cumple con su objetivo, supongamos que un robot choca

contra una pared y el sistema de control calcula la distancia a la pared después del impacto, esos datos ya no son útiles por muy correctos que esten.

Un sistema en tiempo real puede ser de 2 tipos: estricto o *hard real-time* y no estricto o *soft real-time*.

El sistema en tiempo real estricto tiene los requerimientos más rigurosos y garantiza de que la tarea será completada dentro de su período especificado.

El sistema en tiempo real no estricto es menos restrictivo y se limita a garantizar que las tareas de tiempo real críticas tengan prioridad sobre otras tareas y que conserven dicha prioridad hasta completarse.

2.3.1. Implementación de sistemas operativos en tiempo real

Un sistema operativo en tiempo real al ser de único propósito e implementado para sistemas con poca capacidad de hardware, como un sistema embebido en la mayoría de los casos, posee menos

funcionalidades que un sistema operativo común que se dedica a atender múltiples programas, periféricos sofisticados, estrictas políticas de seguridad, interfaces gráficas, etc.

Un sistema operativo en tiempo real se concentra en lo básico e importante que le permita cumplir con su objetivo o propósito: responder dentro del tiempo límite o *deadline*. Para esto se debe enfocar en los procesos y sus prioridades de ejecución en el kernel, minimizar retardos, minimizar interrupciones de periféricos, entre otros.

Teniendo esto en cuenta se identifica las características más importantes para implementar un sistema operativo en tiempo real:

- Apropiación de kernel
- Planificación de procesos
- Latencia minimizada

2.3.1.1. Kernels apropiativos

Al igual que la planificación apropiativa, un kernel apropiativo permite desalojar una tarea que se está ejecutando en modo kernel de ser necesario. Además que puede tener una mejor capacidad de respuesta ya que existe menos riesgo de que un proceso en modo kernel se ejecute durante un periodo de tiempo arbitrariamente largo.

En caso de tener un kernel no apropiativo la tarea en tiempo real tendrá que esperar a que la tarea que se encuentra activa en el kernel termine de ser ejecutada.

Hay diversas estrategias para hacer que el kernel sea apropiativo. Una estrategia puede ser usar puntos de desalojos donde se comprueba si existe la necesidad de ejecutar un proceso de alta prioridad; de ser así se da paso al proceso de alta prioridad y cuando éste

termine se retoma el proceso anteriormente interrumpido. La otra estrategia es el uso de mecanismos de sincronización con señales de espera (wait signal()), en el que la ejecución de un proceso se paraliza momentáneamente y puede dar paso a que otro proceso entre en ejecución.

2.3.1.2. Planificación apropiativa basada en prioridades

Nos valemos de este tipo de algoritmo de planificación que permite darle a los procesos más importantes la prioridad más alta sobre los procesos menos importantes; así un proceso que este en el CPU puede ser desalojado cuando un proceso de mayor prioridad pasa a estar disponible para ser ejecutado. Supongamos un proceso indicando la temperatura del ambiente y otro proceso

indicando el camino a seguir en un robot, este segundo proceso es el que evitará alguna colisión, por lo cual su prioridad de ejecución debería ser mayor frente a la medición de temperatura.

Un planificador apropiativo basado en prioridades sólo garantiza la funcionalidad de los sistemas reales no estrictos, mientras que los estrictos deben garantizar que las tareas en tiempo real reciban servicio de acuerdo a sus requisitos de temporización y para ello necesita otras características de planificación adicionales (ver Sección 2.4.2).

2.3.1.3. Minimización de latencia

Supongamos que una caldera está próxima a llegar a la temperatura máxima para la que fue

diseñada, el sistema debe atenderlo y responder lo más rápido posible.

Ahora, esto no sucede de manera inmediata, ya que transcurre un tiempo entre que se da el suceso y en el que es atendido; a esta diferencia de tiempo se le denomina Latencia de suceso. La Figura 2.3 lo muestra de manera gráfica:

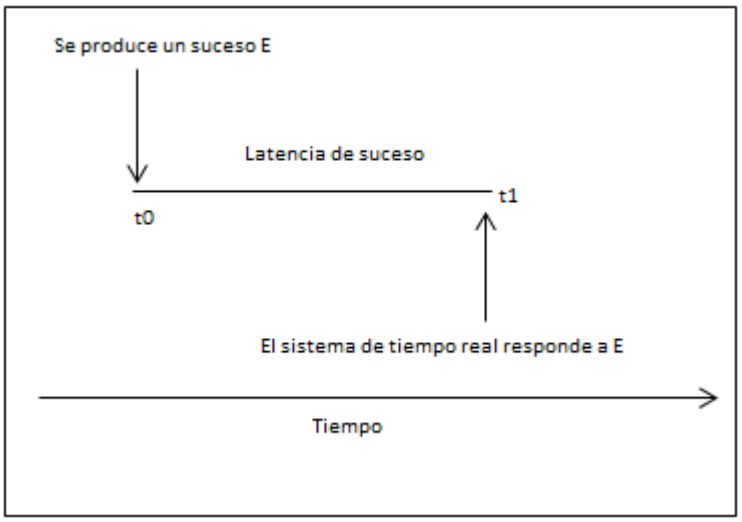


Figura 2.3 Latencia de suceso

Hay dos tipos de latencia que afectan el desempeño de los sistemas en tiempo real:

- Latencia de interrupción

- Latencia de despacho

Latencia de interrupción se refiere al tiempo que pasa entre la llegada de la interrupción a la CPU y el instante en que empieza la atención a dicha interrupción.

El sistema primero debe terminar la tarea que está en ejecución, luego determinar el tipo de interrupción y ahí atender a la interrupción que ha ocurrido. El tiempo que se tarda en seguir estos pasos se denomina latencia de interrupción. Los sistemas de tiempo real deben minimizar esta latencia, o acotarlas como en el caso de un sistema real estricto, al mínimo para poder garantizar que las tareas sean atendidas de manera inmediata.

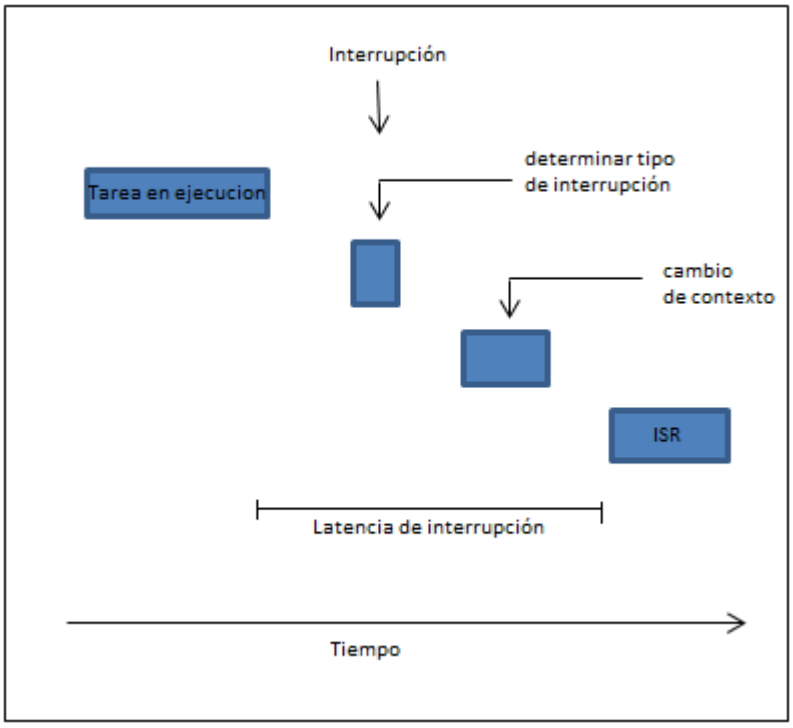


Figura 2.4 Latencia de interrupción

Latencia de despacho se refiere en cambio al tiempo que tarda el despachador del planificador detenga un proceso e inicie otro. Este tiene dos componentes:

- Desalojo de cualquier proceso que se esté ejecutando en el kernel.
- La liberación de los procesos de baja prioridad ocupando recursos para que otros procesos de mayor prioridad los utilice.

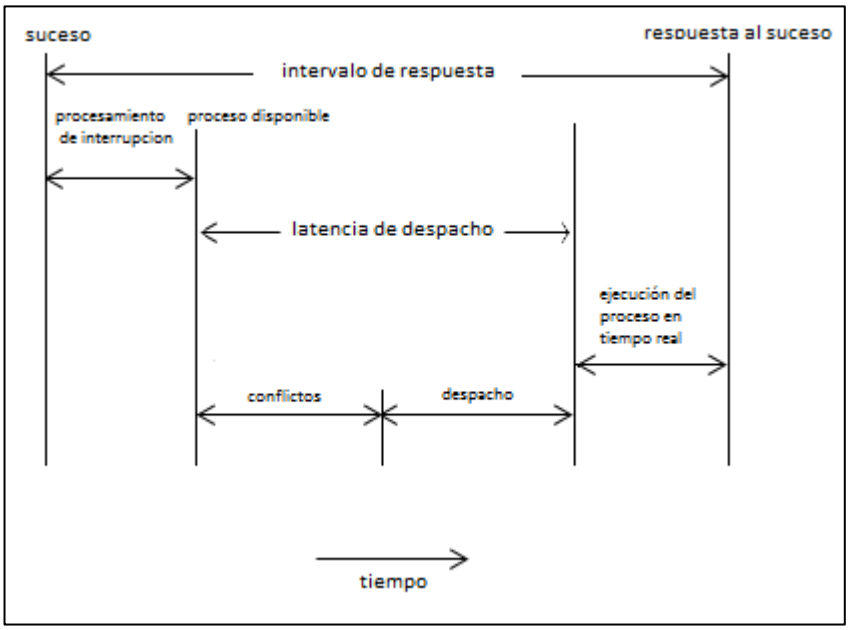


Figura 2.5 Latencia de despacho

2.3.2. Planificación de la CPU en tiempo real

Definamos primero dos características que debemos considerar antes de planificar con este método. Primero, los procesos se consideran periódicos ya que requieren el uso del CPU cada cierto tiempo constante. Segundo, cada proceso periódico tiene un tiempo *t* una vez que se apodera de la CPU, un tiempo *d* que es el máximo tiempo en que debe ser terminado y un período *p*. La relación entre estos 3 factores es:

0 <= t <= d <= p. La tasa de una tarea periódica es 1/p.

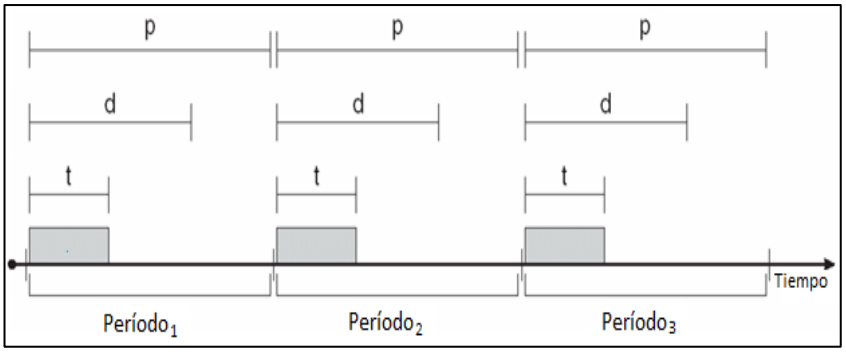


Figura 2.6 Planificación del CPU en tiempo real

Las prioridades son asignadas de acuerdo al período del proceso o de su tiempo límite de ejecución.

Este tipo de planificación obliga a que el proceso anuncie al planificador en uso sus requisitos sobre el tiempo que va a necesitar de servicio. Entonces, el planificador puede aceptar el proceso o rechazarlo según vea si puede cumplir con el plazo que requiere para ser ejecutado o no.

2.3.2.1. Planificación por prioridad monótona en tasa

Este algoritmo planifica tareas periódicas. Consiste en proporcionar prioridades estáticas a los procesos. Si se está ejecutando un proceso de menor prioridad y otro de mayor prioridad pasa a estar disponible para ejecución, este proceso desalojará el proceso de menor prioridad. A cada tarea periódica se le asigna una prioridad inversa a su período: cuanto más pequeño sea su periodo, mayor será su prioridad y viceversa. Con esta política de asignación se logra que los procesos que solicitan al CPU con más frecuencia tengan la mayor prioridad.

Esta tipo de planificación se considera óptima porque si un conjunto de procesos no puede planificarse con este algoritmo, no podrá ser planificado por ningún otro algoritmo que use el mecanismo de prioridades estáticas.

Pero a pesar de ser considerada óptima tiene una limitación y es que no siempre se puede maximizar por completo los recursos del CPU, ya que depende del número de procesos a planificar.

Esta no es la única manera de planificar procesos, como mencionamos en la Sección 2.2 de Planificador existen varias políticas de planificación, por ejemplo: planificación por prioridad en finalización de plazo, planificación por cuota proporcional, planificación en Pthread que consisten en asignar prioridades de acuerdo al plazo de finalización, o asignando tiempo a cada proceso de acuerdo al número de procesos que vayan a ejecutarse, o atenderlos en el orden de llegada, todo esto según como corresponda a cada política. Sin embargo, es esta planificación de tasa monótona la que se selecciona para nuestro proyecto de tesis, ya que contamos con

procesos periódicos en los que conocemos cuál es su tiempo máximo en el que deben ejecutarse y permite que su ejecución sea simultánea.

2.4. RTAI

2.4.1. RTAI

RTAI cuyas siglas provienen de las palabras en inglés Real Time Application Interface, es un conjunto de librerías, o Application Programming Interface (de ahora en adelante API), que permite realizar aplicaciones en tiempo real con estrictas limitaciones de tiempo.

Es un proyecto de software libre desarrollado por el DIAMP, Departamento de ingeniería aeroespacial del Politécnido de Milan (Deparment of Aerospace Engineering of Politecnico di Milano).

2.4.2. Arquitectura

RTAI derivó del proyecto RTLinux [6]. RTLinux es un sistema operativo en tiempo real estricto que consiste en un microkernel de Linux. En dicho microkernel se encuentra un sistema operativo completo de Linux corriendo como una tarea más y de baja prioridad. El microkernel es el que tiene el control evitando que las interrupciones del sistema sean atendidas hasta cuando no haya tareas en tiempo real por realizar.

RTAI no es precisamente un sistema operativo, si no se basa en el kernel de Linux que le permite hacer uso de sus características, servicios y soportes pero teniendo el control sobre las interrupciones habituales del sistema, ciertas abstracciones y facilidades para poder realizar las tareas en tiempo real. Las interrupciones las puede atender el manejador de interrupciones de RTAI o ser pasadas al manejador de

interrupciones del sistema operativo inmerso en el microkernel.

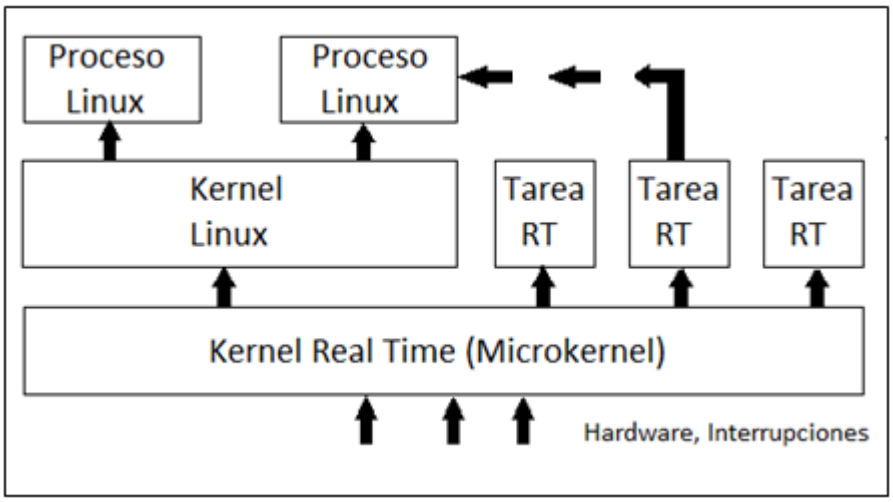


Figura 2.7 Funcionamiento de RTAI

RTAI puede ofrecer como mínimo los siguientes servicios:

- Manejo de hardware e interrupciones de periféricos.
- Planificadores, activación de procesos, manejo de prioridades, temporizadores.
- Comunicación entre aplicaciones.

Los planificadores con los que cuenta RTAI son: uniprocesador (UP), multiuniprocesador (MUP),

Multiprocesador simétrico (SMP), multiprocesador (MP) [7].

En el uniprocesador realiza un algoritmo de planificación para seleccionar la tarea que se va a ejecutar. Cualquier proceso con una alta prioridad puede hacer uso del CPU.

El multiprocesador simétrico está diseñado para máquinas SMP y permite seleccionar qué procesador o procesadores pueden ejecutar la tarea, si no se le especifica él lo asigna según la carga de trabajo del procesador o procesadores en uso.

El multiuniprocesador necesita que se le indique en que procesador ejecutar la tarea de inicio aunque luego ésta tarea pueda ser asignada a otro procesador. Todas las funciones de UP y MUP están disponibles en este planificador.

Al igual que para memoria compartida, compatibilidad POSIX, uso de números flotantes, sincronización de tareas mediante semáforos, exclusión mutua y colas

de mensajes, llamadas a procedimientos remotos (RPCs), buzones e incluso la posibilidad de usar RTAI desde el espacio de usuario.

Puede ser usado desde algunas arquitecturas, como por ejemplo: x86, x86_64, PowerPC, ARM, entre otras [8].

2.4.3. Módulos

Los módulos son archivos que contienen código compilado en lenguaje máquina que permiten extender las funciones del kernel, ya sea un driver de un dispositivo o un módulo que realice llamadas al sistema. Cuando ya no se requiere el uso de un módulo este puede ser “descargado” del kernel liberando memoria o también ser “cargado” sin la necesidad de reiniciar el sistema en cada momento.

En RTAI los módulos que se cargan en el kernel para poder hacer uso de su API son los siguientes:

- lxrt
- rtai

- rtai_sched
- rtai_shm
- rtai_fifos

Al igual que los módulos de cualquier kernel estos pueden ser cargados y descargados utilizando los comandos insmod y rmmod respectivamente.

lxrt proviene de LX(Linux)RT(RealTime), implementa los servicios que hace que los planificadores de RTAI estén disponibles para los procesos de Linux, es decir, que se pueda compartir memoria, enviar mensajes, usar semáforos temporizadores: Linux – Linux, Linux – RTAI, y naturalmente RTAI – RTAI.

rtai es el que ofrece el marco de trabajo principal, inicializa las variables de control y de estructuras. Una vez que se carga este módulo en ese mismo instante el Linux estándar ya no está en poder de habilitar y deshabilitar las interrupciones. Desde ese momento

rtai se asegura que las habilitaciones y deshabilitaciones tengan consistencia.

rtai_sched proporciona la planificación en modo de disparo (o oneshot mode, la tarea se ejecuta en tiempo arbitrarios) y en modo periódico (o periodic mode, la tarea se ejecuta periódicamente). RTAI considera la prioridad 0 como la más alta prioridad y a 0x3fffFfff la más baja prioridad. Este módulo está cargado automáticamente después de *rtai*.

rtai_shm este módulo permite compartir memoria a través de diferentes tareas en tiempo real y los procesos de Linux estándar simultáneamente.

rtai_fifos implementa los servicios fifo (first in first out) para RTAI. A pesar de no ser necesario gracias a lxrt aún se lo conserva al ser muy útil en la comunicación con el manejador de interrupciones.

Estos no son los únicos módulos en RTAI, pero sí los más importantes.

2.4.4. Interfaz de programación de RTAI

Esta sección describe el uso del API de RTAI que podemos encontrar en su página oficial [5] [6].

Incluiremos en este documento solamente las funciones que se han utilizado en nuestros experimentos.

Tipos de datos:

RT_TASK es la estructura para crear el puntero de las tareas en tiempo real

RTIME es un long long y puede almacenar hasta 64 bits

Funciones:

- start_rt_timer(period)

  Arranca el temporizador con el periodo "period". Requerido sólo en modo periódico porque en modo de disparo ese valor es ignorado.

- rt_task_make_periodic (RT_TASK *task, RTIME start_time, RTIME period)

Hace que la tarea corra periódicamente intervalos específicados por el periodo period.

- period: período medido en unidades de ticks de reloj (clock ticks)
- start_time: es el tiempo para la primera ejecución. Medida en ticks de reloj y a partir del tiempo actual.

- rt_set_oneshot_mode()

  Hace que la tarea corra en modo de disparo, es decir, la tarea sea ejecutada arbitrariamente. Debe ir antes de cualquier función que tenga que ver con temporizadores o relacionadas con el tiempo incluyendo las conversiones como nano2count entre otras.

- void rt_task_wait_period(void)

  Espera al siguiente periodo. La tarea es suspendida temporalmente mientras cede el control hasta que el próximo período de tiempo es alcanzado.

- nano2count(RTIME nanosecs)

  Convierte loa nanosegundos en unidades de ticks del reloj.

- rt_get_time()

  Retorna el número de ticks del reloj que ha pasado desde que el temporizador en tiempo real arrancó. Este número es múltiplo de la frecuencia proporcionada por un PIT (Programmable Interval Timer) 8254 (1193180 Hz) en modo periódico y es múltiplo del periodo del reloj del CPU en modo de disparo.

- rt_get_cpu_time_ns()

  Igual que la función rt_get_time solo que esta retorna en nanosegundos los ticks del reloj.

- rt_sem_init(SEM *sem, int value)

  Inicializa una pila FIFO de semáforos -Un semáforo puede ser usado para comunicación y sincronización a través de tareas en tiempo real. *sem* es un puntero a la estructura SEM y *value*

es el valor inicial del semáforo. Los semáforos en RTAI asumen que su contador nunca va a exceder 0xFFFF que es la señal que se usa para retornar un error al igual que el valor inicial tampoco puede ser 0xFFFF.

- rt_sem_broadcast(SEM* sem)

  Desbloquea a todas las tareas que están en espera.

- rt_receive(RT_TASK* task, unsigned int *msg)

  Recibe un mensaje de la tarea *task*. Si *task* es igual a 0, quien llama acepta mensajes de cualquier tarea. Si hay una tarea pendiente, rt_receive no bloquea pero se puede anticipar si la tarea rt_sent fue recibida por el mensaje tiene una prioridad mayor.

- rt_send(RT_TASK* task, unsigned int msg)

  Envía un mensaje *msg* a la tarea *task*. Si quien recibe está lista para el mensaje rt_send no

bloquea a la tarea que envía, pero su ejecución puede ser anticipada por una tarea receptora de mayor prioridad. De otro modo la tarea es bloqueada y empilada en el orden de prioridades en la lista de recibir de la tarea que envía.

- rt_sem_signal(SEM* sem)

  Señaliza un evento o un semáforo. Es llamado cuando una tarea deja una sección crítica. El valor del semáforo es incrementado y probado. Si el valor no es positivo, la primera tarea en la cola de espera del semáforo tiene permiso de continuar. Nunca bloquea la tarea desde donde es llamado. Retorna 0 si todo estuvo bien o retorna un valor negativo si hubo alguna falla.

- rt_sem_wait(SEM* sem)

  Toma un semáforo. Espera hasta un evento que se señale a un semáforo. Es llamado cuando una tarea entra a una sección crítica. El valor del semáforo es decrementado y probado. Si aún no

es negativo rt_sem_wait retorna inmediatamente. Caso contrario la tarea es bloqueada y encolada. El encolamiento puede ser en orden de prioridad o basado en FIFO. Esto es determinado por tiempo de compilación SEM_PRIORD. En este caso rt_sem_wait retorna cuando la tarea que llama está primer lugar de la cola de espera y otra tarea emite una llamada o un error ocurre, por ejemplo el semáforo es destruido. Retorna el número de eventos que ya se señalizaron exitosamente.

- rt_task_init_schmod(RT_TASK *task, int priority, int stack_size, int max_msg_size, int policy, int cpus_allowed)

  Crea una tarea pero con uso del planificador.

  - task: puntero a la tarea en tiempo real.
  - priority: prioridad que le asignaremos a la tarea. 0 es la mayor prioridad para RTAI y

la menor prioridad se le obtiene con RT_LOWEST_ PRIORITY.

- stack_size: tamaño de la pila para ser usado en la tarea
- max_msg_size: el máximo tamaño del mensaje a utilizar en la tarea.
- policy: para indicar qué tipo de planificador vamos a usar: FIFO, Round Robin o algún otro planificador. Se indica con SCHED_FIFO, SCHED_RR o SCHED_OTHER que es el planificador por defecto.
- cpus_allowed: los cpus permitidos -0 para policy y cpus_allowed es para indicar los valores por defecto.

En los anexos se encuentran los códigos de los experimentos realizados.

2.4.5. Codificación de un programa en RTAI

En esta sección se explica cómo hacer un pequeño programa en espacio de usuario con RTAI, usaremos para ilustrar uno de los programas desarrollado para una de nuestras pruebas codificado en lenguaje C.

El programa muestra el promedio de los tiempos obtenidos del CPU y el máximo valor después de realizar un número de consultas del tiempo al CPU; se la realiza en sentido estricto y no estricto.

1.- Incluimos las librerías que vamos a necesitar para las funciones que tendrá nuestro programa, sobre todo las de RTAI.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sched.h>
```

--Estas son las librerías para RTAI:

```
#include <rtai_sem.h>
#include <rtai_msg.h>
#include "period.h"
```

--Definiciones de algunas variables si es necesario

```
#define ONE_SHOT
#define NO_OF_ITERATIONS        1000000
#define TASKBASE 1000
```

2.- Dentro de nuestro bloque principal hacemos las declaraciones de las variables que vamos a necesitar

```
int main(void)
{
        RT_TASK *mytask;
        unsigned long mytask_name;
        int i, count;
        RTIME times[NO_OF_ITERATIONS];
        RTIME total,delta,maximum;
        struct sched_param mysched;
        mytask_name = TASKBASE;
```

--La prioridad que vamos a manejar para nuestra tarea es la mínima que me puede asignar el planificador FIFO.

```
mysched.sched_priority = sched_get_priority_min(SCHED_FIFO);
```

3.- Indicamos el tipo de planificador

```
if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
        printf("ERROR IN SETTING THE POSIX SCHEDULER\n");
        exit(1);
}

mlockall(MCL_CURRENT | MCL_FUTURE);
```

4.- Iniciamos nuestra tarea en tiempo real

```
if (!(mytask = rt_task_init(mytask_name, 1, 0, 0))) {
        printf("CANNOT INIT TASK %lu\n", mytask_name);
        exit(1);
}
printf("TASK INIT: name = %lu, address = %p.\n",
mytask_name, mytask);

printf("soft[0] o hard[1]\n");
scanf("%d",&i);
```

5.- Indicamos si nuestra tarea va a ser periódica o no periodica. En nuestro caso no es periódica sino de modo de disparo, es decir, va a hacer ejecutada en tiempos arbitrarios.

```
rt_set_oneshot_mode();
```

6.- Iniciamos el temporizador. Sólo debe ser iniciado una sola vez en el programa, otro llamado haría que lo sobreescriba.

-- En modo de disparo el temporizador es ignorado, pero es estrictamente necesaria si nuestra tarea va a ser periódica. Nuestra tarea no es periódica, pero la colocamos para indicar cómo se inicializa el temporizador.

```
start_rt_timer(0);
```

7.- Indicamos si nuestra tarea va a ser en tiempo real estricto o no estricto

```
if (i==0)
        rt_make_soft_real_time();
else
        rt_make_hard_real_time();
```

8.- El resto de lo que se tendría que hacer para cumplir con el objetivo de nuestro programa.

```
for( count = 0; count < NO_OF_ITERATIONS; count++ )
        {
                times[count]=rt_get_cpu_time_ns();
        }

        total = maximum = delta = 0;
        for( count = 0; count < NO_OF_ITERATIONS - 1; count ++ ) {
                delta = (times[count+1] - times[count]);
                total += delta;
                if( delta > maximum )
                {
                maximum = delta;
                }
        }
        printf("The average time for the get_cpu_time() call:
        %.8f\n",(float)(total/ count)/10e9);
        printf("The maximum time for the get_cpu_time() call:
        %.8f\n", (float)maximum/10e9);
```

9.- Detenemos el temporizador.

```
stop_rt_timer();
```

10.- Eliminamos el puntero a la tarea.

```
rt_task_delete(mytask);
```

```
return(0);
```

Los pasos del 1 al 10 son los básicos para realizar un programa.

2.4.6. Ejecutar un programa en RTAI

Para poder ejecutar necesitamos compilar nuestro programa y cargar los módulos de RTAI conforme se necesite. Los módulos se deben cargar en un orden específico, ya que existe dependencia entre los módulos, se lo puede ver con claridad en el Anexo A, que trata la instalación de RTAI y se explica cómo verificar si fue realizado con éxito. Para facilitar estos pasos lo más recomendable es utilizar los scripts que vienen en la carpeta del showroom [12] que contiene ejemplos de programas utilizando RTAI para modo usuario y modo kernel.

Para ejecutar el programa nos valemos de los scripts anteriormente mencionado y que están en la sección de anexos.

Primero, ejecutamos el MAKEFILE [Anexo G] que es quien me compilará mi código y me dará como resultado mi programa ejecutable. Luego, corremos nuestro script RUN [Anexo H] que levanta los módulos

de RTAI y busca mi archivo ejecutable que corresponde al programa.

Ahora, los scripts, el código y la librería *period.h* no pueden estar en cualquier lugar ya que así no funcionarían los scripts. Deben estar en cierto orden:

- Tenemos la carpeta que contiene nuestro programa y los scripts. Una subcarpeta donde irá: el código del programa, la librería *period.h*, *makefile*, *run* y *rem*.
- Fuera de la subcarpeta irá: *ldmod* y *remod*.

Teniendo en cuenta esta sugerencia todo debería poder ejecutarse sin problema alguno.

2.4.7. Python con RTAI

De la misma manera cómo podemos realizar programas en C utilizando las herramientas de RTAI podemos realizar programas con el lenguaje Python como el programa de la Sección 2.4.6, que también fue codificado en lenguaje Python [Anexo D].

En el código se notan pequeñas diferencias que tenemos que tomar en cuenta al querer usar Python.

- En el paso 1 la forma en cómo se hacen las importaciones de las librerías de RTAI.

```
From rtai_lxrt import *
```

- Python no hace declaraciones de variables, pero sí se puede tener variables previamente definidas.
- Se indica de igual manera el planificador que se vaya a usar. En las pruebas que realizamos utilizando Python-RTAI usamos la función:

```
mytask=rt_task_init_schmod(mytask_name,1,0,0,0,0xF)
```

Crea la tarea en tiempo real e indica el planificador a usar. En el API se puede hacer una lectura más profunda de esta función.

- De los pasos 6 al 10 que describen cómo realizar un programa en RTAI sigue de la misma forma secuencial.

Así como para ejecutar los programas en C con RTAI hacemos usos de scripts igualmente con los programas en Python y las recomendaciones son las mismas con respecto a la ubicación de ellos.

CAPITULO 3

3. HERRAMIENTAS

3.1. Introducción

Este capítulo se centra en el uso de las herramientas que nos ayudarán a cumplir nuestros objetivos.

Definiremos el propósito de cada herramienta para crear un punto de partida al estudio de los resultados obtenidos.

Comenzaremos con Open Plataform Communications (OPC), su uso inicial y como al ser utilizado como plataforma de comunicación para los instrumentos simulados en nuestro sistema, esta sección cubrirá su definición, arquitectura y propósito como herramienta de comunicación.

3.2. Open Plataform Communications

La herramienta de comunicación que utilizamos en nuestras pruebas es Open Platform Communications (OPC) nombrada así en el 2011 por la OPC Foundation [8], que es un Consorcio de Industrias que manejan estándares.

OPC es una especificación de un protocolo de comunicaciones de tiempo real que crea un interface común para instrumentos de distinto fabricante, como válvulas de control, PLCs y demás dispositivos de medición y por ende con distintos controladores de software, puedan interactuar y compartir datos.

Este protocolo implementa la tecnología OLE, que es sistema de objetos distribuido y un protocolo desarrollado por Microsoft que se usa para transferir datos entre aplicaciones esto junto a otras tecnologías de comunicación como COM y DCOM que son tambien plataformas de Microsoft nos permitirán trabajar implementando objetos neutrales, con

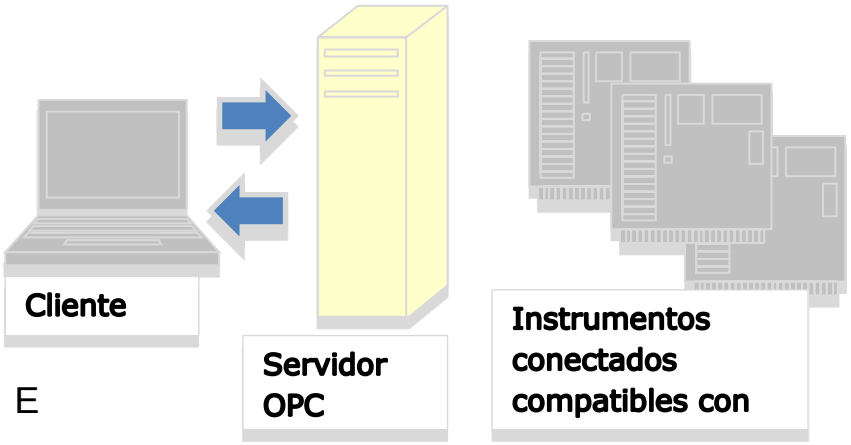
respecto al lenguaje que se use para programar, así podremos crear objetos que manipulen los datos de instrumentos de control para nuestro llegar a nuestro objetivo.

3.2.1. Arquitectura OPC

Un Servidor OPC es un software que "lee", conoce o interpreta el lenguaje propietario del hardware o software para obtener datos de diferentes dispositivos compatibles.

Un Cliente OPC es un computador que hace peticiones de ciertos datos de interés al Servidor OPC para poder después monitorearlos o manipularlos si es necesario. Como se puede ver en la Figura 3.1, la arquitectura básica sería muy parecida a esto.

Este Sistema OPC mantiene una arquitectura común entre todas las configuración leídas y consultadas a expertos en esta área para nuestro estudio.



E

Figura 3.1 Arquitectura básica de un sistema OPC

Nosotros comparamos los procesos de lectura de datos con diferentes protocolos enfrentándolos a pruebas a cada uno para poder mostrar cómo podría aprovecharse las técnicas de programación en tiempo real, para crear un cliente que pueda tener acceso a datos de un servidor de manera más óptima posible.

Advosol [9] muestra una comparativa entre OPC DA y XML-DA, pero aquí básicamente definiremos las principales diferencias entre estos dos protocolos

Primero OPC DA es el más utilizado por ser el primero en ser estandarizado implementando Objetos DCOM que soporta todo sistema en plataforma Microsoft

Windows. En cambio XML-DA basa su especificación en estándares web como XML, SOAP y WSDL y estandariza los mensajes SOAP entre clientes y servidores.

Ambas tienen ventajas sobre la otra pero en nuestro estudio comparamos las lecturas bajo ciertas condiciones y los tiempos de respuesta que estos nos han mostrado.

La creación estándar de un cliente OPC no varía con respecto a la tecnología de protocolo a usar, se maneja de la siguiente forma en una tabla demostrativa [10] [11].

- Creación de una instancia del Servidor almacenada en un objeto.

OPC DA (OpenOPC)	XML-DA (PyOPC)
opc = OpenOPC.client() o opc = OpenOPC.open_client(address)	xda = XDAClient (OPCServerAddress=address , 10 ReturnErrorText=True)

Conectarse a un servidor.

OPC DA (OpenOPC)	XML-DA (PyOPC)
opc.connect('Matrikon.OPC.Simulation')	xda = XDAClient (OPCServerAddress= address , 10 ReturnErrorText=True)

- Leer datos de ese servidor

OPC DA (OpenOPC)	XML-DA (PyOPC)
opc.read('Random.Int4') (19169, 'Good', '06/24/07 15:56:11')	printoptions(xda.Read([ItemContainer(ItemName='simpleitem' ,MaxAge=500)],16LocaleID='en−us'))

3.3. Soporte de lenguajes de programación en RTAI

Como indicamos anteriormente, entre los Sistemas en tiempo real, una de nuestras herramientas más poderosas fue RTAI.

RTAI tiene soporte para lenguajes C y Python, al tener esta posibilidad nosotros utilizaremos ambos en el sistema de Tiempo Real.

Lenguaje C es un lenguaje compilado de alto soporte de librerías en tiempo real c, y como es fuertemente tipificado nos permite crear aplicaciones robustas que en nuestro caso nos permitieron realizar los experimentos de rendimiento de las herramientas de RTAI comparadas contra las herramientas comunes de trabajo, con pruebas de rendimiento utilizando llamadas al sistema.

Python es un lenguaje interpretado y trabaja por medio de scripts, este nos permite crear diferentes funciones dependiendo del uso y objetivo que tenemos para nuestras pruebas, al ser interpretado y no compilado carece de manejo avanzado de hilos y semáforos a diferencia de la versatilidad que nos ofrece Lenguaje C.

En nuestro capítulo de experimentos podremos analizar cómo se desempeñan estas dos herramientas realizando pruebas comparativas, bajo ciertas

condiciones que demuestren sus ventajas en nuestros estudios.

CAPITULO 4

4. RESULTADOS

4.1. Experimento # 1. Justificación del uso de RTAI

Evaluar la diferencia del tiempo de respuesta de un sistema desarrollado con herramienta RTAI versus realizarlo con el kernel estándar Linux.

4.1.1. Descripción del experimento

Se realizan las llamadas al sistema en el API de Linux estándar y usando las herramientas de RTAI para un sistema de tiempo real en sentido estricto y no estricto, en este capítulo *hard realtime* y *soft realtime* respectivamente.

La prueba se realiza libre de interrupciones por parte de la interfaz gráfica, ya que es deshabilitada al igual que las conexiones de red.

La versión de kernel tanto para el Linux estándar y RTAI es 2.6.24.7.

El programa realiza las llamadas al sistema a través de las funciones gettimeofday y rt_get_cpu_time_ns para el API de Linux estándar y kernel de RTAI respectivamente. Ambas funciones obtienen el tiempo actual del CPU en tics de reloj en nanosegundos. Se guardan los tiempos de las llamadas y se realizan los cómputos para obtener un promedio de las llamadas y el máximo de los tiempos obtenidos en cada ejecución. El programa fue ejecutado 5 veces con un número de 1000000 llamadas en cada ejecución.

Los códigos correspondientes a esta prueba se encuentran en el anexo B y C.

La Figura 4.1 muestra los tiempos promedios obtenidos.

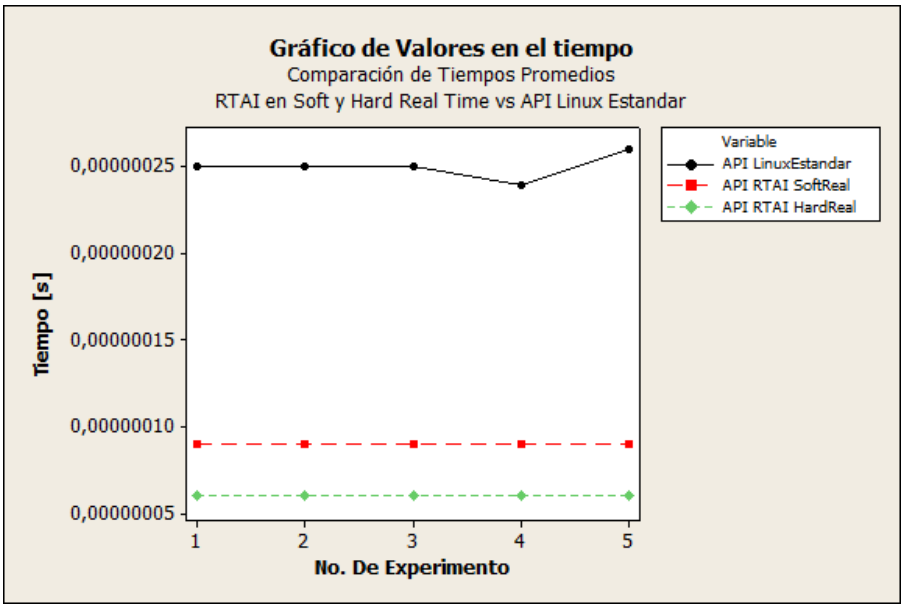


Figura 4.1 Tiempos promedios API Estándar vs API RTAI en Soft y Hard realtime

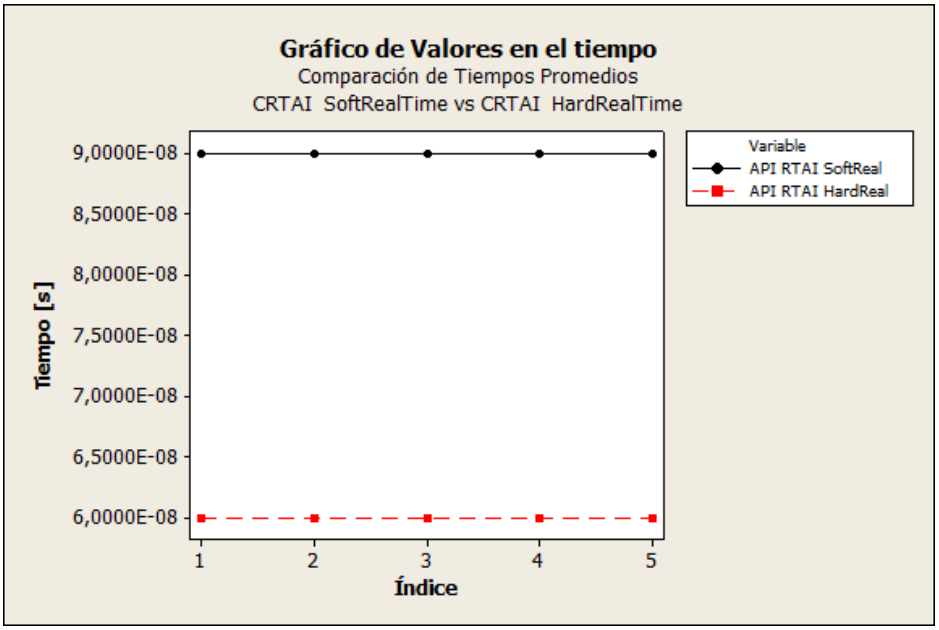


Figura 4.2 Tiempos promedios API RTAI Soft realtime versus Hard realtime

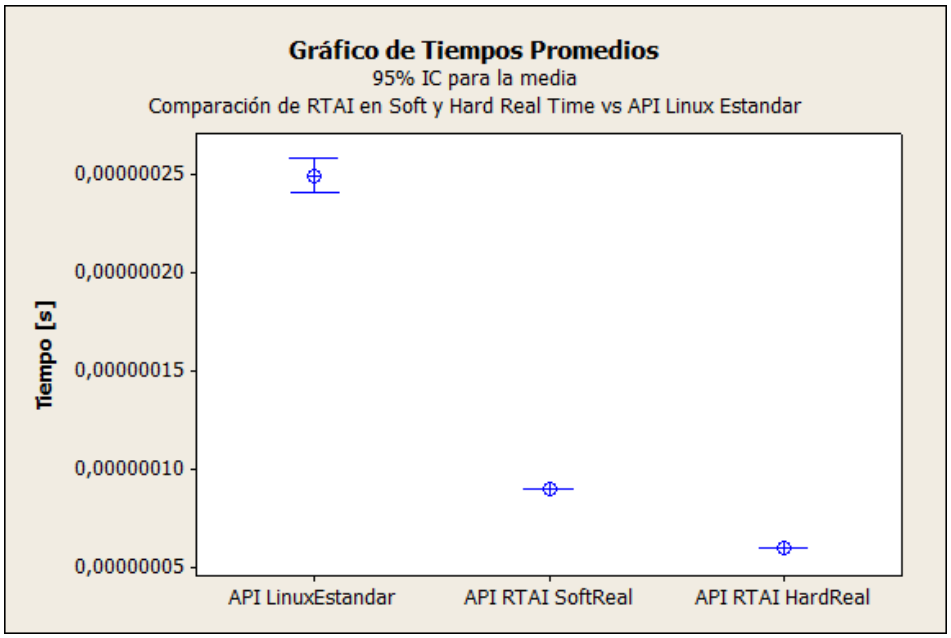


Figure 4.3 Varianza de tiempo promedio API Estándar, API RTAI en soft y hard realtime

4.1.2. Resultados obtenidos

La Tabla 4.1 resume los promedios del experimento realizado.

	API ESTÁNDAR	API RTAI SOFT REALTIME	API RTAI HARD REALTIME
PROMEDIO [ns]	250	90	60
σ (MÁXIMO)	1.918E-05	4.08718E-07	5.47723E-09

Tabla 4.1

Tenemos el promedio de las llamadas al sistema (PROMEDIO) y la desviación estándar de los máximos obtenidos (MÁXIMO) de todas las ejecuciones.

Como resultado obtuvimos que el kernel estándar Linux para las llamadas al sistema responden aproximadamente 4 veces más lento que usando RTAI. Sin embargo, si hablamos del promedio de sus valores máximos es 7 veces mayor que en soft realtime y 66 veces mayor que en hard realtime. Así mismo, en hard realtime y soft realtime la diferencia en respuesta es mínima, mgas en sus máximos notamos que en hard realtime es aproximadamente 9 veces menor. Podemos notar que hard realtime es 1.5 veces más rápido que soft realtime.

De la Figura 4.1 notamos que no hay mucha variación en los resultados tanto para el API ESTÁNDAR como para soft y hard realtime con RTAI en sus promedios.

De la Figura 4.2 podemos observar que los valores tanto para soft y hard realtime se mantienen constantes, e incluso que entre ellos punto a punto la diferencia se mantiene en aproxidamente 1.5 veces

siendo, como es de esperarse, hard realtime el de mayor rapidez al responder.

4.2. Experimento # 2. Comparar rapidez de respuesta entre Python-RTAI Hard Realtime y Python-RTAI Soft Realtime

En este experimento se comparan los tiempos promedios empleados por una tarea en tiempo real con programada en lenguaje Python con herramientas RTAI en hard realtime contra la misma tarea en soft realtime.

4.2.1. Descripción del experimento

El programa realiza las llamadas al sistema a través de las funciones rt_get_cpu_time_ns en hard realtime y soft realtime. Se obtiene el tiempo del CPU del sistema en nanosegundos. Se guardan los tiempos de las llamadas y se realizan los cómputos para obtener un promedio de las llamadas y el máximo de los tiempos obtenidos en cada ejecución. El programa

fue ejecutado 5 veces en para hard y soft realtime con un número de 1000000 llamadas en cada ejecución.

Los códigos correspondientes a esta prueba se encuentran en el Anexo D. La Figura 4.4 nos muestra los promedios obtenidos.

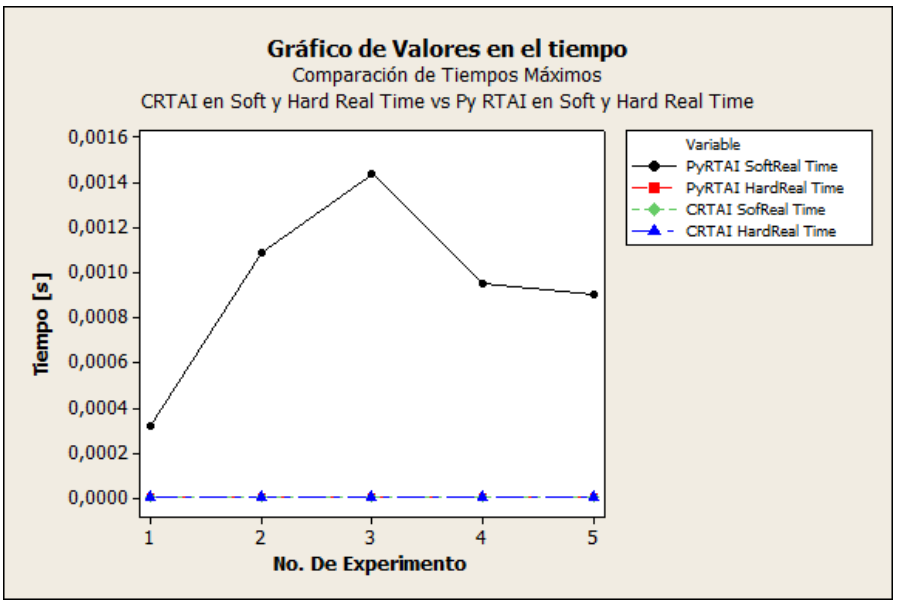


Figura 4.4 Tiempo promedio Python-Rtai en Soft realtime versus Hard realtime

4.2.2. Resultados obtenidos

A continuación presentamos el resumen de los datos.

	Hard Realtime	Soft Realtime
Promedios	190 ns	260 ns
σ (MÁXIMO)	6.5491E-08	4.0482E-04

Tabla 4.2

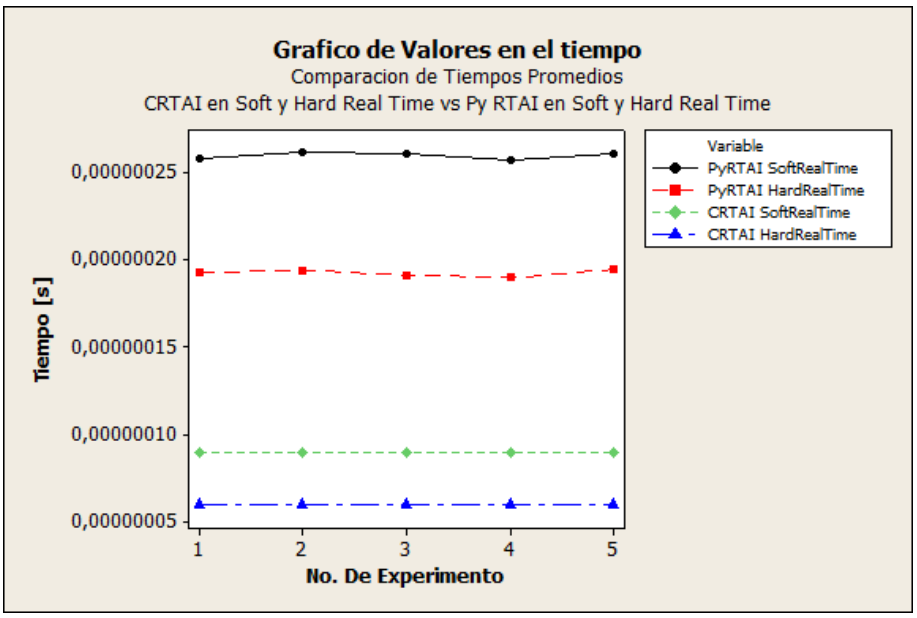
En la Figura 4.4 observamos que en hard realtime tenemos en cada ejecución valores que son prácticamente los mismos, lo que le da predictibilidad al programa mientras que en soft realtime sí observamos variación aunque no sean muy distantes unos resultados con otros.

En promedio los valores máximos de soft realtime sí se encuentran distantes uno de otros mientras que en hard realtime se mantiene constante.

De la tabla podemos deducir que un sistema programado en Python con herramientas RTAI y en hard realtime va a responder aproximadamente 1.35 veces más rápido que uno en soft realtime.

4.3. Comparación C-RTAI vs Python-RTAI

Esta sección tiene como objetivo comparar los tiempos de una tarea en tiempo real con un programa escrito en Python-RTAI versus una tarea en tiempo real escrito con un programa C-RTAI. Nos basamos en los experimentos #1 y #2 detallados en la Sección 4



? **Figura 4.5 C-RTAI versus Python-RTAI con Hard y Soft Realtime**

.

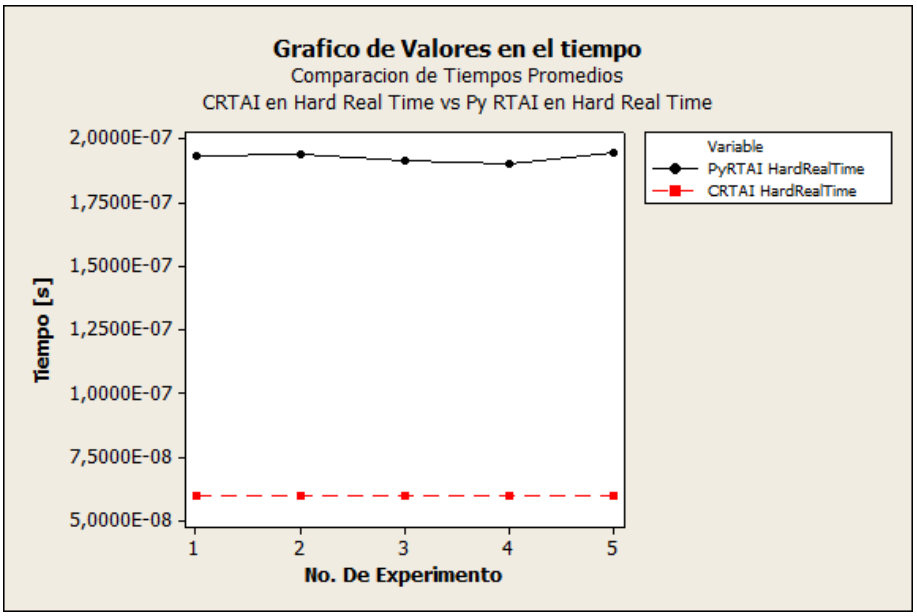


Figure 4.6 C-RTAI Hard Realtime versus Python-RTAI Hard Realtime

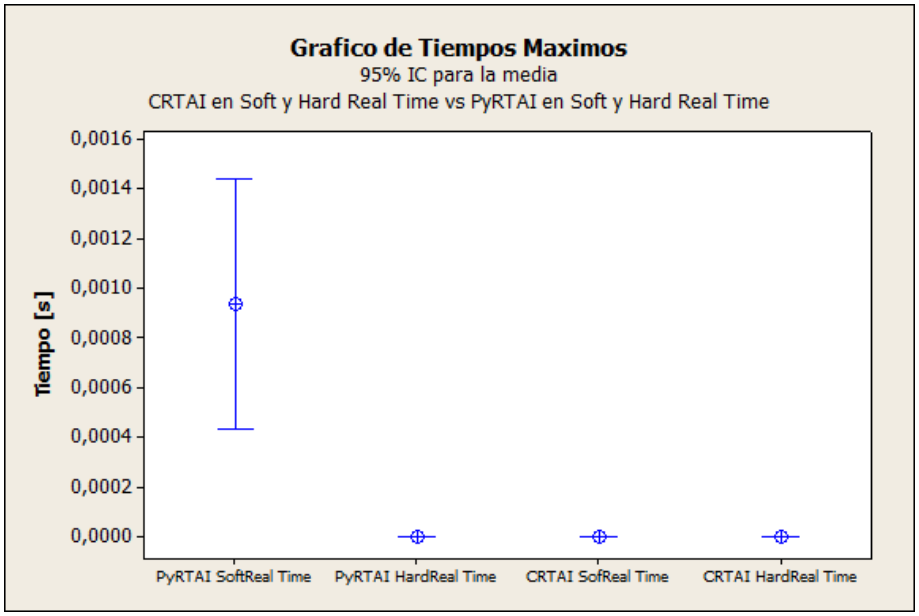


Figure 4.7 Varianzas de tiempos promedios C-RTAI versus Python-RTAI en Hard y Soft Realtime

	PyRTAI Soft Realtime	PyRTAI Hard Realtime	C-RTAI Soft Realtime	C-RTAI Hard Realtime
Promedio	260 [ns]	190 [ns]	90 [ns]	60 [ns]

Tabla 4.3

4.3.1. Resultados obtenidos

En la Figura 4.5 observamos una diferencia considerable entre los resultados del experimento al ser realizado con C-RTAI en soft y hard realtime versus las pruebas realizadas con el mismo programa en Python-RTAI. Diferencia de que soft realtime en C-RTAI es 1.5 veces más lento que en hard realtime con C-RTAI, pero 2.9 y 2 veces más rápido que soft y hard realtime en Python-RTAI respectivamente.

Con respecto a hard realtime en C-RTAI versus Python-RTAI observamos que el primero responde 3 veces más rápido.

Como habíamos notado en las observaciones de los experimentos de la Sección 4.1 y 4.2, las diferencias que hay entre las pruebas realizadas en hard realtime que difieren en el lenguaje Python-RTAI y C-RTAI, ambas mantienen constancia en el tiempo e incluso la diferencia entre ellos se mantiene en cada uno de los experimentos realizados.

4.4. Experimento # 3. Comparar rapidez de lectura entre los protocolos XML-DA y OPC DA

En este experimento se busca definir que herramientas que implementen el protocolo XML-DA u OPC DA va a ser utilizado.

4.4.1. Descripción del experimento

Para esta prueba se tiene 2 servidores: uno comunicándose bajo el protocolo XML-DA y el otro bajo OPC DA. El servidor XML-DA se encuentra instalado en Ubuntu y el servidor OPC-DA en Windows 7.

Se cuenta con sus respectivos clientes, ambos implementados bajo las herramientas de RTAI en soft realtime.

Servidor y cliente se encuentran conectados directamente con cable de red simulando una LAN dedicada.

La prueba consiste en realizar 100 lecturas de datos de cada servidor y tomar el tiempo que tarda en realizarlas.

En los Anexos E y F encontramos los respectivos programas para esta prueba.

La Figura 4.8 a continuación muestra el resultado de haber realizado la prueba 5 veces por cada servidor.

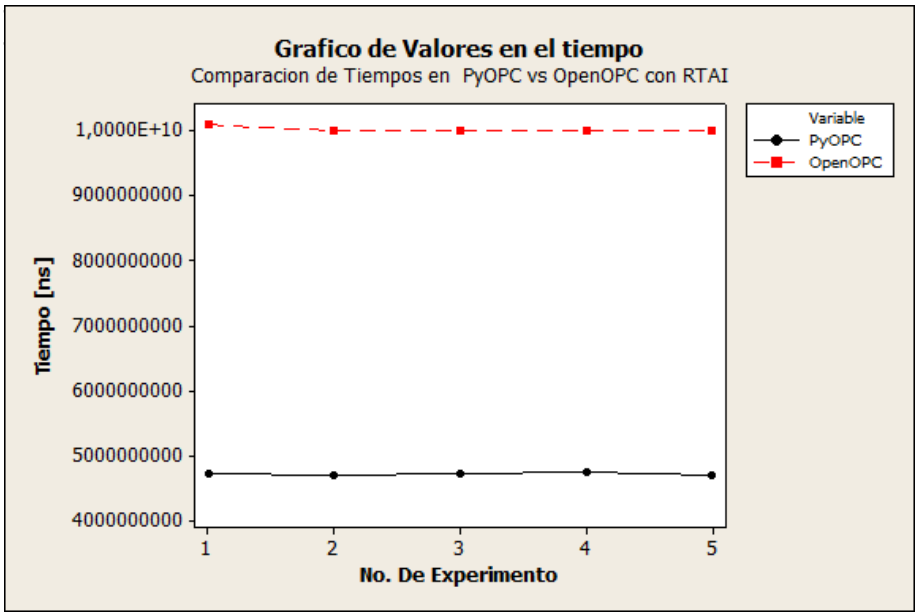


Figura 4.8 Tiempos de lectura PyOCP versus OpenOPC

4.4.2. Resultados obtenidos

De la Figura 4.8 podemos ver la diferencia de tiempos entre ambos protocolos, siendo PyOPC el que más rápido responde en comparación a OpenOPC. Notamos también que ambos protocolos son constantes en el tiempo de respuesta, es decir, ninguna petición responde en menos o más tiempo que otra.

A continuación el promedio del tiempo que transcurría al realizar las 100 lecturas en cada una de las 5 ejecuciones:

	PyOPC	**OpenOPC**
100 lecturas	4.72 [s]	10 [s]
1 lectura	0.0472 [s]	0.1 [s]

Tabla 4.4

OpenOPC tarda aproximadamente el doble en responder que PyOPC.

4.5. Experimento # 4. Comparar rapidez de respuesta con tráfico de red del protocolo XML-DA y OPC DA

4.5.1. Descripción del experimento

Para esta prueba se tiene 2 servidores: uno comunicándose bajo el protocolo XML-DA y el otro bajo OPC DA. Se cuenta con sus respectivos clientes, ambos implementados bajo las herramientas de RTAI.

La prueba consiste en realizar 100 lecturas de datos de cada servidor y tomar el tiempo que tarda en realizarlas, pero con un tráfico de red simulado en 1000000 paquetes que se trasmiten en la red dedicada para comunicación de cliente- servidor. La red dedicada la conseguimos conectando directamente el cliente y servidor mediante un cable de Ethernet.

En los Anexos E y F se encuentran los respectivos programas para esta prueba.

La Figura 4.9 a continuación nos muestra los tiempos obtenidos bajo las condiciones expuestas.

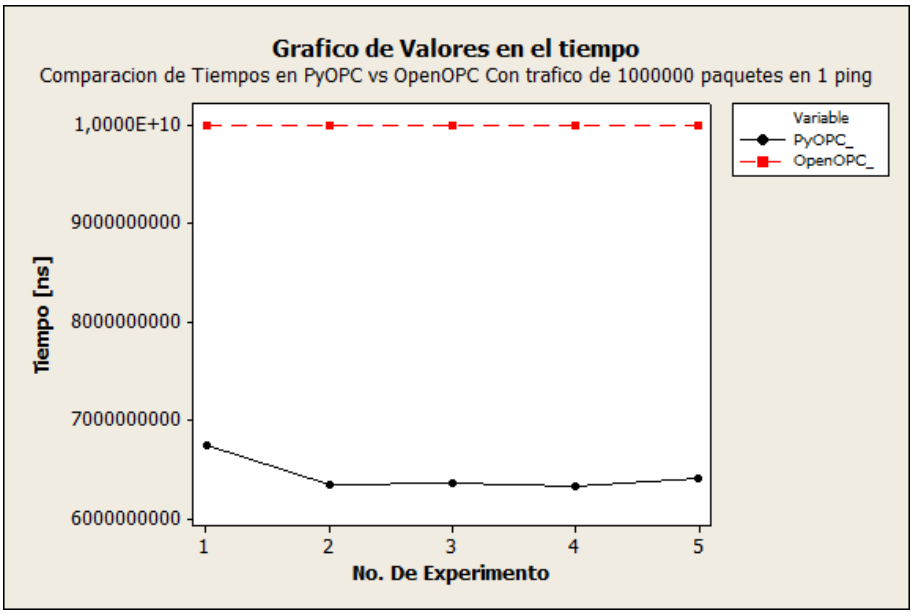


Figura 4.9 Tiempos de lectura con tráfico de red PyOPC versus OpenOPC

4.5.2. Resultados obtenidos

Podemos observar de la Figura 4.9 que sigue manteniéndose la diferencia en tiempo entre ambos protocolos al igual que en el experimento de la Sección 4.4.

A continuación el promedio del tiempo que transcurría al realizar las 100 lecturas en cada una de las 5 ejecuciones con tráfico de red:

	PyOPC	OpenOPC
100 lecturas	6.43 [s]	9.99 [s]
1 lectura	0.0643 [s]	0.099 [s]

Tabla 4.5

OpenOPC no se vió afectado por el tráfico de red, ya que su tiempo de respuesta se mantuvo igual al experimento sin tráfico de red, PyOPC aumentó 1.36 veces el tiempo de respuesta frente al tráfico de red y en este caso OpenOPC tarda 1.55 veces en responder que PyOPC.

4.6. Experimento #5: Planificación en tasa monótonica de la comunicación de varios servidores, implementados bajo el protocolo XML-DA, como procesos haciendo uso de las herramientas RTAI

Este es el experimento final de nuestra tesis que integra todos los resultados y análisis de nuestros experimentos anteriores utilizando técnica de planificación de procesos.

Fue realizado con 2 servidores y posteriormente se indica hasta cuántos servidores en línea podemos

tener. En la Limitaciones [Sección 4.7] y Recomendaciones [Sección 4.8] comentaremos las limitaciones con las que nos encontramos a lo largo del desarrollo, así como las recomendaciones para futuras investigaciones.

4.6.1. Descripción del experimento

Los servidores son simulados con la herramienta PyOPC. El experimento fue realizado con 2 servidores y un cliente.

Los servidores se encuentran ejecutando en una misma PC con Ubuntu 8.04. Cada servidor está enviando sus datos a través de un puerto, por ejemplo, un servidor puede estar por el puerto 8000 y el otro por el 8001 a pesar de que están compartiendo la misma IP.

El cliente se encuentra en otra PC que contiene el parche de RTAI, igualmente en Ubuntu 8.04.

La conexión servidor-cliente es mediante un cable Ethernet simulando una red dedicada que estos servidores utilizan en la industria.

En el Anexo I se encuentra el código escrito en lenguaje Python con herramientas RTAI que se ejecuta desde el cliente que permite establecer la conexión con el servidor como la lectura de sus datos.

Creamos un hilo (thread) por cada servidor, cada uno de ellos está en hard realtime, establece la conexión con el servidor y va a realizar las lecturas. Las tareas serán periódicas, mientras una de ellas espera la otra adquiere datos.

La planificación de cada uno de los procesos, que serían cada uno de los hilos relacionados con los servidores, se realizó en base a la política de Planificación por prioridad monótona en tasa [Sección 2.4.2.1], en la cual necesitamos un período p, un tiempo de lectura t, un tiempo límite d. Estos valores *t* y *d* fueron obtenidos en los experimentos #3 y #4 en la

Sección 4.3 y 4.4, en los que tomamos los tiempos de lectura con y sin tráfico de red, *p* va a depender del número de servidores que estemos monitoreando por el valor del tiempo límite quedando estas variables definidas así:

t=0.0472 [ns], d=0.0643 [ns], p=#servidores*d

Donde *t* es el tiempo que tarda en realizar una lectura el cliente al servidor, *d* es el tiempo que tarda en la lectura con tráfico de red y que nos servirá también para el período p. Decidimos que el periodo debería ser el valor del tiempo límite por el número de servidores para que en un período se puedan hacer una lectura de cada servidor para no perder tiempo del CPU sin utilizar y porque al realizar nuestra prueba final están directamente conectado los servidores sin tráfico de red, entonces no deberían llegar a sus tiempo límite sin haber realizado la lectura correspondiente.

Los procesos son periódicos para que un servidor realice su lectura y permita que el siguiente servidor haga uso del servidor mientras espera que su periodo se cumpla.

El diagrama de planificación se encuentra a continuación:

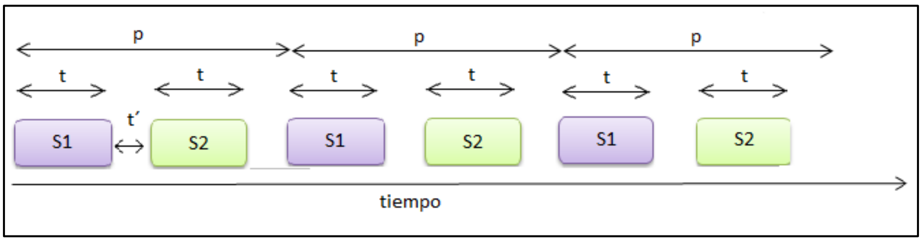


Figura 4.10 Esquema de planificación de la comunicación de los servidores

t´ es el tiempo que tarda en retomar la lectura del siguiente servidor que es de 0.02 [s].

En las Figuras 4.11 y 4.12 mostramos cómo se presentan las lecturas de los servidores:

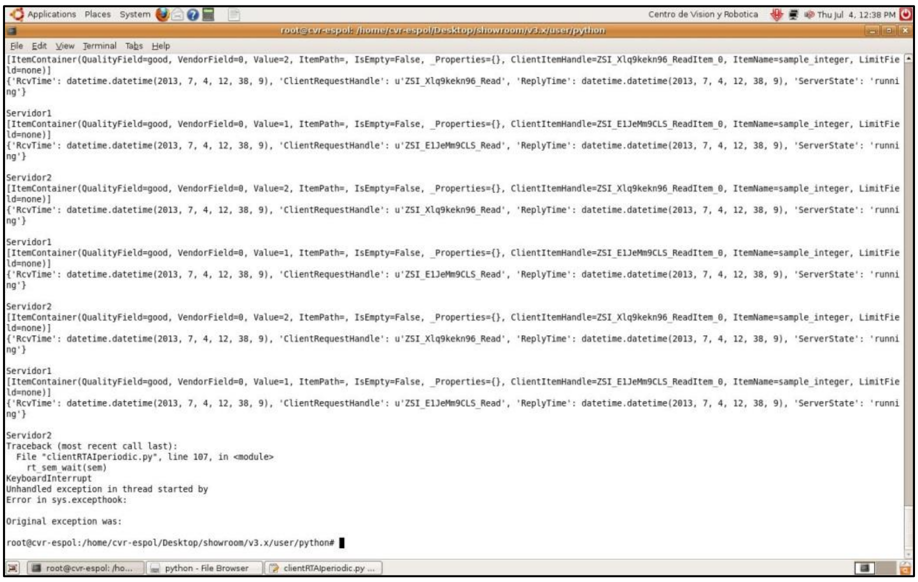


Figura 4.11 Lectura de servidores

En la Figura 4.12 a continuación muestra el manejo de error que simula si un servidor deja de funcionar:

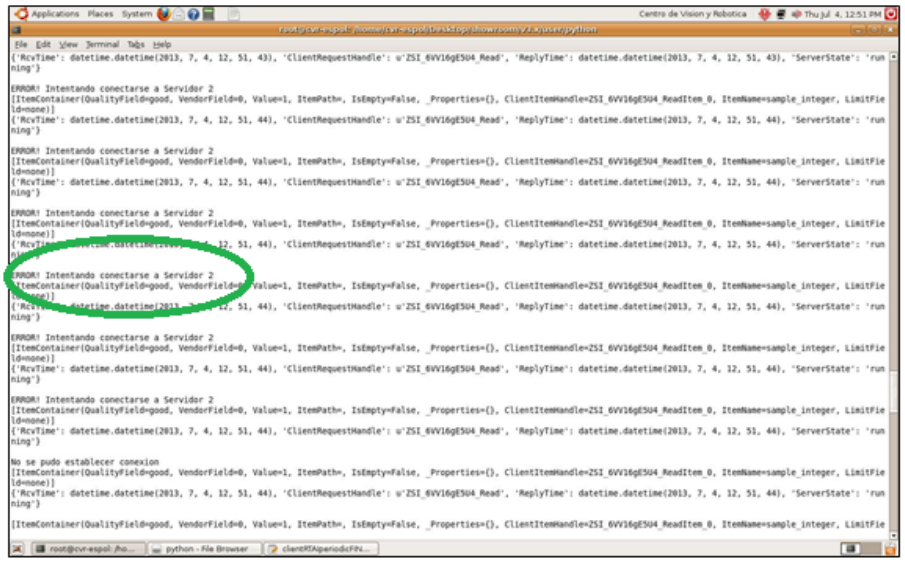


Figure 4.12 Manejo de error en comunicación de servidores

la Figura 4.13 demuestra qué sucedería en caso de que las tareas no fueran periódicas, se ha omitido que imprima los valores obtenidos del servidor para una mejor muestra:

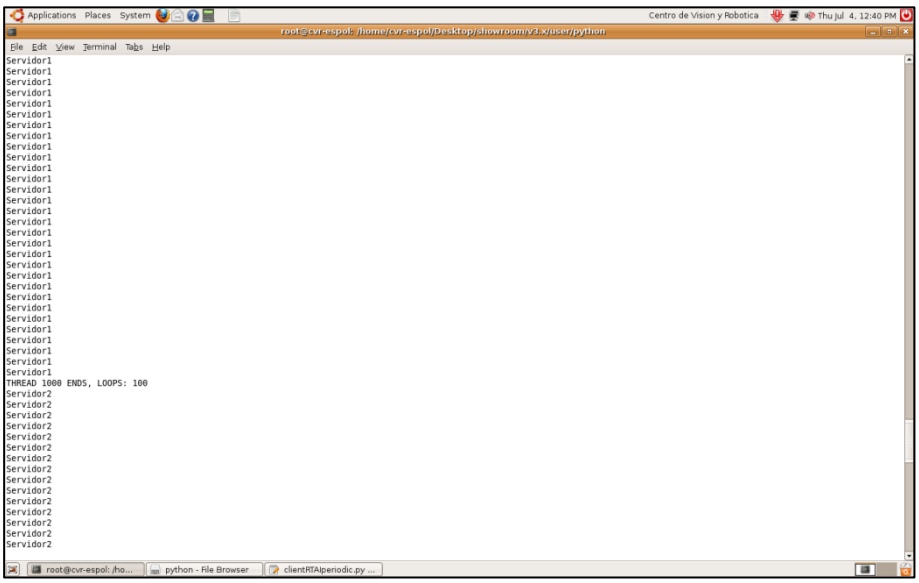


Figure 4.13 Comunicación no periódica

4.6.2. Resultados Obtenidos

En las Figuras 4.11, 4.12 y 4.13 tenemos los resultados finales de nuestra última prueba.

Observando la Figura 4.11 vemos que se hace la lectura de un servidor y luego del otro cumpliendo con la planificación y la lectura simultánea.

En la Figura 4.12 tenemos los mensajes de error al haberse “desconectado” un servidor que en nuestro caso es dejar de ejecutar el servidor.

Por último en la Figura 4.13 tenemos la muestra de que pasaría si no hubieran sido periódicas las tareas en tiempo real que se encargan de hacer las lecturas de los datos de los servidores. Como vemos, primero haría las lecturas de un servidor y una vez que haya terminado daría paso para las lecturas del siguiente servidor. Esto sucede porque al ser tareas hard realtime ellas se apoderan del uso del CPU que no abandonan hasta terminar con su ejecución, pero al

ser periódicas se ven en la obligación de esperar su siguiente turno.

Con esto demostramos: el uso de las herramientas RTAI con aplicación en el monitoreo de control.

LIMITACIONES

1. Los servidores son simulados.
2. Los servidores al ser simulados sus datos no pueden ser cambiados mientras se esté ejecutando porque son una instancia que se marca desde un inicio. Si quiere que dé otro valor hay que cancelar la ejecución y volverlo a ejecutar.
3. Todos los servidores se encuentra ejecutando en la misma PC aunque en diferentes puertos, es decir: 8080, 8081, etc.
4. El tiempo de lectura fue tomado con la ejecución de un solo servidor, por eso al querer ejecutar más de 2 servidores ya se pierde simultaneidad.

5. El tráfico de red fue simulado con 1 millón de pings desde el servidor al cliente.
6. Para poder hacer uso del simulador PyOPC hay que importar unas librerías que estaban escritas en lenguaje Python, lo que nos llevó a diseñar el cliente también en el mismo lenguaje.
7. Python a pesar de contar con soporte RTAI no tenía todas las funciones que se puede hacer uso desde C, un manejador de señales por ejemplo (handler).

RECOMENDACIONES

1. Tener servidores en diferentes PCs y en el cliente tener varias tarjetas de red, una por servidor.
2. Realizar lecturas con servidores de diferentes tipos de protocolo, es decir, uno PyOPC y otro OpenOPC al mismo tiempo.
3. Buscar una solución que permita hacer uso del lenguaje C.
4. Tener un hilo que permita controlar el tiempo máximo que debería tomar una lectura en un servidor.

CONCLUSIONES

En esta sección presentaremos las conclusiones que se obtuvieron a lo largo de la realización de todos los experimentos.

1. Hay mayor ventaja en usar el API de RTAI versus un API ESTÁNDAR, ya que me brinda más control sobre los resultados al ser constantes y responde con mayor rapidez.
2. El servidor implementado bajo el protocolo XML-DA responde en la mitad del tiempo que el implementado con protocolo OPC-DA.
3. Ambos protocolos son constantes en su tiempo de respuesta, lo cual los hace “predecibles” a la hora

de saber cuánto tiempo tardará en arrojar un dato frente a una petición.

4. Aún con tráfico de red un servidor implementado con XML-DA es 40% más rápido en responder que uno implementado con OPC DA.
5. Un servidor implementado con OPC DA mantiene su constancia de respuesta frente al tráfico de red.
6. Un servidor XML-DA es susceptible al tráfico de red, pierde constancia en su tiempo de respuesta.
7. Incluso programando con Python un sistema hard realtime tiene la ventaja de ser estable en sus resultados volviéndolo predecible y controlable.
8. La rapidez depende del desempeño del lenguaje empleado, sea Python-RTAI o C-RTAI.
9. Desarrollar en base a herramientas orientados a hard realtime sea en C-RTAI o Python-RTAI facilita que respondan en el menor tiempo que cualquier otra técnica de programación en realtime.

10. El lenguaje de programación implementado, si bien tienen distintas mediciones de rapidez, su diferencia es constante la mayoría del tiempo, dándonos la oportunidad de poder predecir los valores de hacerlo de con un lenguaje versus hacerlo utilizando otro.
11. El lenguaje C aún con herramientas de RTAI se mantiene superior en rapidez de respuesta a Python con herramientas RTAI.

ANEXOS

Anexo A: Instalación de RTAI

La instalación la hicimos bajo la distribución de Ubuntu 8.04, versión de kernel vainilla Linux-2.6.24.7 y versión de RTAI 3.7

Tabla de Dependencias Previas

Cada paquete detallado a continuación debe ser instalado como súper usuario, estando en la terminal desde cualquier carpeta del sistema.

General	**su**	
Software	**Comando**	**Dependencias**
cvs	apt-get install cvs	ninguna
svn	apt-get install subversion	libapr1 libaprutil1 libpq5 libsvn1
build-essential	apt-get install build-essential	build-essential dpkg-dev g++ g++-4.2 libc6-dev libstdc++6-4.2-dev libtimedate-perl linux-libc-dev patch
checkinstall	apt-get install checkinstall	ninguna
Kernel		
Software	**Comando**	**Dependencias**

libncurses5-dev	apt-get install libncurses5-dev	ninguna
kernel-package	apt-get install kernel-package	intltool-debian kernel-package po-debconf
fakeroot	apt-get install fakeroot	ninguna
Rtai		
Software	**Comando**	**Dependencias**
libxmu-dev	apt-get install libxmu-dev	libxmu-dev libxmu-headers
libxi-dev	apt-get install libxi-dev	ninguna
doxygen	apt-get install doxygen	ninguna
autoconf	apt-get install autoconf	ninguna
automake	apt-get install automake	autoconf automake autotools-dev m4
libtool	apt-get install libtool	ninguna

Configuración del Kernel

-Cambiar al directorio /usr/src/

cd /usr/src/

-Descomprimir el archivo rtai-3.6-cv.tar.bz2

tar xjvf rtai-3.7-cv.tar.bz2

-Creamos un vínculo para facilitar el acceso a la carpeta que produjo la anterior descompresión. Nombre de vínculo: rtai

ln -s rtai-3.7-cv rtai

-Descomprimir el Kernel descargado en el paso anterior

tar xjvf linux-2.6.24.7.tar.bz2

-Es recomendable cambiar el nombre de la carpeta que se creó con el paso anterior, para poder identificar fácilmente al kernel en el que vamos a trabajar.

mv /usr/src/linux-2.6.24.7 /usr/src/linux-2.6.24.7-rtai

-Crear un vínculo para un fácil acceso a la carpeta. El nombre del vínculo es linux

ln -s linux-2.6.24.7-rtai linux

-Ingresamos a la carpeta del kernel

cd /usr/src/linux

-Parchar la versión del kernel descargado con el parche ubicado dentro de las carpetas de Rtai.

patch -p1 < /usr/src/rtai/base/arch/x86/patches/hal-linux-2.6.24-x86-2.0-07.patch

-Copiar el archivo de configuración del kernel que ya está corriendo en nuestro sistema y que fue levantado automáticamente durante la instalación de Ubuntu.

cp /boot/config-2.6.24-24-generic .config

-Configurar el nuevo archivo de configuración del Kernel para adecuarlo con las instrucciones del archivo de configuración del Kernel levantado por Ubuntu

make oldconfig

-Levantar la configuración necesaria para adecuar al nuevo Kernel de Linux como Kernel de sistema en tiempo real

make menuconfig

Este comando mostrará en la terminal un menú como se indica en la siguiente figura que nos permitirá elegir los parámetros que ayuden a ajustar nuestro kernel para que realice las tareas en tiempo real.

Parámetros de Configuración del Kernel

Se debe verificar el tipo de procesador y los dispositivos que posee el computador en el cual se va a realizar la instalación para elegir los correctos parámetros, caso contrario la instalación no se realizará con éxito.

Los siguientes comandos le permitirá realizar la recomendaciones anteriormente descritas:

lspci : Lista los dispositivos pci del computador.

cat /proc/cpuinfo : Muestra la información del procesador.

General setup

-Local version - append to kernel release: -rtai

Enable loadable module support

-Module versioning support : deshabilitado

-Source checksum for all modules: habilitado

Processor type and features

-Subarchitecture Type: PC-compatible

-Proccesor family: Seleccionar el procesador que posee el computador

-Preemption Model: Preemptible Kernel (Low-Latency Desktop)

-Interrupt pipeline : habilitado

-Toshiba Laptop support: deshabilitado

-Dell Laptop support: deshabilitado

-High Memory Support: off

-64 bit Memory and IO resources (EXPERIMENTAL): deshabilitado para pc 32 bits

---

Power Management support: deshabilitado

Bus Options (PCI etc.)

-PCCard (PCMCIA/cardBus) support : deshabilitado

Networking

-Networking options

-The IPV6 protocol: deshabilitado

-The DCCP Protocol (Experimental): deshabilitado

-The TIPC Protocol (Experimental): deshabilitado

-Asynchronous Transfer Mode (ATM) (Experimental): deshabilitado

-The IPX Protocol: deshabilitado

-Appletalk protocol support: deshabilitado

-WAN router: deshabilitado

-Amateur Radio support: deshabilitado

-IrDA (infrared) subsystem support: deshabilitado

-Bluetooth subsystem support: deshabilitado

-RxRPC session sockets: deshabilitado

-Wireless: deshabilitar todo el contenido interno

-RF switch subsystem support: deshabilitado

-Plan 9 Resource Sharing Support (9P2000)(Experimental): deshabilitado.

Device Drivers

-Memory Technology Device (MTD) support: deshabilitado

-Parallel port support: deshabilitado

-Misc devices:deshabilitado

-I2O device support: deshabilitado

-Macintosh device drivers :deshabilitado

-Network device support

    -Token Ring driver support: deshabilitado

    -Wireless Lan: deshabilitar todo el contenido interno

    -USB Network Adapters: deshabilitar todo el contenido interno

    -FDDI: deshabilitado

-ISDN support: deshabilitado

-Telephony support: deshabilitado

-Input device support

    -Joystick interface: deshabilitar

    -Joysticks/Gamepads: deshabilitado

    -Tablets: deshabilitado

    -Touchscreens:deshabilitado

-I2C support:deshabilitar todo el contenido interno

-Multimedia devices

    -Video for linux

        Radio Adapters: deshabilitado

    -DAB adapters:deshabilitado

-Graphics support

-/dev/agpart (AGP Support) : habilitado

-Direct Rendering Manager (XFree86 4.1.0 and higher ...: habilitado

-Lowlevel video output switch controls: deshabilitado

-Support for frame buffer devices: deshabilitado

-Backlight & LCD device support: deseleccionar todo el contenido interno

-Display device support: deseleccionar todo el contenido interno

-Console display driver support: deseleccionar todo el contenido interno

-Sound

-Sound card support: deshabilitado

-USB support

-USB Modem (CDC ACM) support: deshabilitado

-USB Printer support: deshabilitado

-Siemens ID USB Mouse Fingerprint sensor support: deshabilitado

-Apple Cinema Display support: deshabilitado

-USB Serial Converter support: deshabilitado

-USB DSL modem support: deshabilitado

-Virtualization: deshabilitado

File systems

-Second extended fs support: habilitado

-Ext3 journalling file system support: habilitado

-Ext3 extended attributes: habilitado

-Reiserfs support: deshabilitado

-CD-ROM/DVD Filesystems: seleccionar todo el contenido interno

- Luego de seleccionar los parámetros construimos los paquetes del Kernel nuevo con los siguientes comando:

make

-Instalar los módulos levantados por el nuevo Kernel

make modules_install

-Instalar los paquetes levantados por el nuevo Kernel

make install

-Copiar el firmware del kernel original con el nombre del nuevo Kernel

cp -aR /lib/firmware/2.6.24-24-generic /lib/firmware/2.6.24.7-rtai

-Crear la imagen de arranque del nuevo kernel

update-initramfs -c -k 2.6.24.7-rtai

-Editar el archivo de configuración del gestor de arranque

gedit /boot/grub/menu.lst

-Colocar el nuevo kernel en las opciones de arranque, con su respectiva maquina e imagen

-Reiniciar el computador y seleccionar el nuevo kernel en el gestor de arranque.

Instalación Rtai parte 1

En esta sección realizamos la instalación de RTAI, lo que nos permitirá ejecutar tareas en tiempo real.

-Ir a la carpeta /usr/src/rtai/

cd /usr/src/rtai

-Verificar los valores del menú de Rtai

-Ejecutar el menú de la consola de configuración de Rtai

make menuconfig

-Generar los archivos configurados en el paso anterior

make

-Realizar la instalación de la configuración actual de Rtai

make install

-Incluir la ruta /usr/realtime/bin en la variable global PATH

echo "PATH=$PATH:/usr/realtime/bin" >> /root/.bashrc

-Incluir la exportación del paso anterior en el script de arranque

echo "export PATH" >> /root/.bashrc

-Reiniciar el computador y seleccionar el kernel de RTAI

Prueba del Progreso de la instalación de RTAI

-Levantar los modulos de la rtai

insmod /usr/realtime/modules/rtai_hal.ko

insmod /usr/realtime/modules/rtai_up.ko

insmod /usr/realtime/modules/rtai_fifos.ko

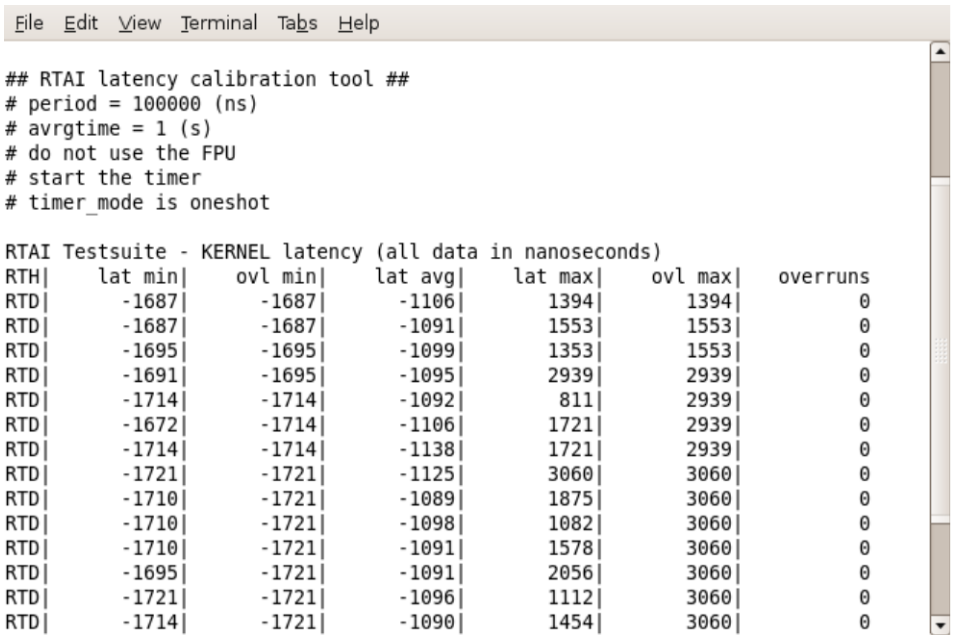
- Ir a la carpeta /usr/realtime/testsuite/kern/latency

cd /usr/realtime/testsuite/kern/latency

-Ejecutar el programa medidor de latencia

./run

Luego de escribir la línea anterior debemos obtener algo similar a como se muestra en la siguiente figura:



Anexo B: Código de experimento # 1

timeCalcRtai.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sched.h>
#include <rtai_sem.h>
#include <rtai_msg.h>
#include "period.h"

#define ONE_SHOT

#define NO_OF_ITERATIONS       1000000
#define TASKBASE 1000

int main(void)
{
        RT_TASK *mytask;
        unsigned long mytask_name;
        int i, count;
        RTIME times[NO_OF_ITERATIONS];
        RTIME total,delta,maximum;
```

```
        struct sched_param mysched;

        mytask_name = TASKBASE;

        mysched.sched_priority = sched_get_priority_min(SCHED_FIFO);
       if( sched_setscheduler( 0, SCHED_FIFO, &mysched ) == -1 ) {
                printf("ERROR IN SETTING THE POSIX SCHEDULER\n");
                exit(1);
        }
        mlockall(MCL_CURRENT | MCL_FUTURE);

        if (!(mytask = rt_task_init(mytask_name, 1, 0, 0))) {
                printf("CANNOT INIT TASK %lu\n", mytask_name);
                exit(1);
        }
        printf("TASK INIT: name = %lu, address = %p.\n", mytask_name, mytask);
        printf("soft[0] o hard[1]\n");
        scanf("%d",&i);
       // start_rt_timer(0);
        rt_set_oneshot_mode();
       start_rt_timer(0);
        if (i==0)
                rt_make_soft_real_time();
       else
                rt_make_hard_real_time();
```

```
        for( count = 0; count < NO_OF_ITERATIONS; count++ )
        {
                times[count]=rt_get_cpu_time_ns();
                //times[count]=rt_get_time_ns();
        }

        total = maximum = delta = 0;
        for( count = 0; count < NO_OF_ITERATIONS - 1; count ++ )
{
                /* Calculate the seconds part of the delta between
                count and count + 1 */
                delta = (times[count+1] - times[count]);
                /* Add the delta to the total for calculating the average
*/
                total += delta;
                /* If delta greater than the maximum, replace the
maximum */
                if( delta > maximum )
                {
                maximum = delta;
                }
        }

         printf( "The maximum time for the get_cpu_time()
        call: %lld\n", maximum );*/

         printf( "The average time for the get_cpu_time()
        call:  %.8f\n",(float)(total / count)/10e9);
         printf( "The maximum time for the get_cpu_time()
```

```
        call:  %.8f\n", (float)maximum/10e9);

       stop_rt_timer();
        rt_task_delete(mytask);
return(0);
}
```

Anexo C: Código de experimento # 1

timeCalc.c

```
/*
*  * Needed Includes
*   */
#define _GNU_SOURCE
#include <sys/time.h>          /* gettimeofday() */
#include <unistd.h>            /* gettimeofday() */
#include <stdio.h>             /* printf() */
#include <sched.h>
#include <stdlib.h>
#include <sys/syscall.h>

/*
*    * NO_OF_ITERATIONS is the number of times that the gettimeofday() call will
*    * be called.  This number is large to allow for a large statistical set from
*     * which the average time can be acquired.
*     */
#define NO_OF_ITERATIONS       1000000
/*
*   * MICRO_PER_SECOND defines the number of microseconds in a second
*   */
#define MICRO_PER_SECOND    1000000
/*
*  * Start of the Main Program
*   */
int main( int argc, char *argv[] )
{
/*
```

```
* * times array holds the timeval structure returned by each subsequent
* * call to gettimeofday().  These values are stored in this array and
* * held for later analysis.
* */
struct timeval times[NO_OF_ITERATIONS];
/*
*         * Other important variables
*                 */
float total;
float delta;
float maximum;
int count;
/**Mask del cpu set para carga **/
cpu_set_t mask;

        /**Afinidad*/
        CPU_ZERO( &mask );
        CPU_SET(0, &mask );
        sched_setaffinity( 0, sizeof(mask), &mask );
/*
** First, loop NO_OF_ITERATIONS times and each time through the loop
* * make a call to gettimeofday() and store the results in the times
* * array.
* */
for( count = 0; count < NO_OF_ITERATIONS; count++ )
{
        gettimeofday( &times[count], NULL );
}
/*
```

```
* * Second, perform a second loop to analysis the results.  NOTE that
* * this loop depends on a pair of time values so it is only performed
* * NO_OF_ITERATIONS - 1 times.
* */
total = maximum = delta = 0;
for( count = 0; count < NO_OF_ITERATIONS - 1; count ++ ) {
        /* Calculate the seconds part of the delta between count and
        count + 1 */
        delta  = times[count+1].tv_sec  - times[count].tv_sec;
        /* Calculate the microseconds part of the delta between
        count and count + 1 */
        delta+=(times[count+1].tv_usec                    -
times[count].tv_usec)/(float)MICRO_PER_SECOND;
        /* Add the delta to the total for calculating the average */
        total += delta;
        /* If delta greater than the maximum, replace the maximum */
        if( delta > maximum )
        {
        maximum = delta;
        }
}
/*
** Third, display the results,  NOTE:  the results are only displayed to
* * the microsecond precision.  This is because the gettimeofday() call
* * is only accurate to microseconds and any value beyond that is
* * "garbage."
**/
printf( "The average time for the gettimeofday() call: %.8f\n", total /
count );
```

```
printf( "The maximum time for the gettimeofday() call: %.8f\n", maximum );
}
```

Anexo D: Código de experimento # 2

```
#!/usr/bin/python

from rtai_lxrt import *

from rtai_msg  import *

from rtai_sem import *

from math import *

from array import *

from thread import *

from random import *

printf = libc.printf


NITERATIONS = 1000000

TASKBASE = 1000

times=(c_longlong * NITERATIONS)()


#main

mytask_name = TASKBASE


mytask = rt_task_init_schmod(mytask_name, 1, 0, 0, 0, 0xF)
```

```
if mytask == NULL :
        printf("CANNOT INIT TASK %lu\n", mytask_name)

printf("TASK INIT: name = %lu, address = %p.\n", mytask_name,
mytask)
rt_set_oneshot_mode()
start_rt_timer(0)
n = int(input("soft[0] o hard[1]\n"))
if (n==0):
        rt_make_soft_real_time()
else :
        rt_make_hard_real_time()
#Guardamos los tiempos obtenidos en el arreglo NITERATIONS
veces
for count in range(0, NITERATIONS):
        times[count]=(c_longlong(rt_get_cpu_time_ns())).value

total=delta=maximum=0
```

```
for count in range(0, len(times) - 1) :
        # Calculate the seconds part of the delta between count and
        count + #1
        delta = (times[count+1] - times[count])
        # Add the delta to the total for calculating the average */
        total =total+ delta

        # If delta greater than the maximum, replace the maximum */
        if( delta > maximum ):
             maximum = delta

average=(total/count)/10e9
maximum=maximum/10e9

print ("The average time for the get_cpu_time() call:\n", average)
print ("The maximum time for the get_cpu_time()
call:\n",maximum)
stop_rt_timer()
rt_task_delete(mytask)
```

Anexo E: Código de experimento # 3

```
#!/usr/bin/env python
#Librerias RTAI
from rtai_lxrt import *
from rtai_msg  import *
from rtai_sem import *
#Librerias PyOPC
from PyOPC.OPCContainers import *
from PyOPC.XDAClient import XDAClient
printf = libc.printf

LOOPS = 100
def print_options((ilist,options)):
    print ilist; print options; print

#main
mytask_name = nam2num("MASTER")
mytask = rt_task_init_schmod(mytask_name, 1, 0, 0, 0, 0xF)
if mytask == NULL :
```

```
        printf("CANNOT INIT TASK %lu\n", mytask_name)


printf ("MASTER INIT: name = %lu, address = %p.\n",
mytask_name, mytask)
start_rt_timer(0)
#address='http://violin.qwer.tk:8000/'
address='http://200.126.1.165:8000/'
xda = XDAClient(OPCServerAddress=address,
                ReturnErrorText=True)
print_options(xda.GetStatus())
print_options(xda.Browse())
count=0
tss = rt_get_cpu_time_ns()
while count < LOOPS :
        count += 1
        print_options(xda.Read([ItemContainer(ItemName='sample_i
        nteger', MaxAge=500)],  LocaleID='en-us'))
tss = rt_get_cpu_time_ns() - tss
print tss
```

```
stop_rt_timer()

#end
```

ANEXO F: Código de experimento # 3

```
#!/usr/bin/env python

#Librerias RTAI


from rtai_lxrt import *

from rtai_msg  import *

from rtai_sem import *

import sys


sys.path.insert(0, 'OpenOPC/src/')


import OpenOPC

printf = libc.printf


LOOPS = 100

mytask_name = nam2num("MASTER")


mytask = rt_task_init_schmod(mytask_name, 1, 0, 0, 0, 0xF)

if mytask == NULL :
```

```
        printf("CANNOT INIT TASK %lu\n", mytask_name)

printf ("MASTER INIT: name = %lu, address = %p.\n",
mytask_name, mytask)

address='192.168.1.2'
start_rt_timer(0)
opc = OpenOPC.open_client(address)
opc.connect('Matrikon.OPC.Simulation')
count=0
tss = rt_get_cpu_time_ns()
while count < LOOPS:
        count += 1
        print opc.read('Random.Int4')

tss = rt_get_cpu_time_ns() - tss
print tss
stop_rt_timer()
```

ANEXO G: MAKEFILE

MAKEFILE

```
prefix := $(shell rtai-config --prefix)

ifeq ($(prefix),)
$(error Please add <rtai-install>/bin to your PATH variable)
endif

CC = $(shell rtai-config --cc)
LXRT_CFLAGS = $(shell rtai-config --lxrt-cflags)
LXRT_LDFLAGS = $(shell rtai-config --lxrt-ldflags)

all: timeCalcRtai

timeCalcRtai: timeCalcRtai.c period.h
	$(CC) $(LXRT_CFLAGS) -o $@ $< $(LXRT_LDFLAGS)
```

```
clean:
	rm -f *.o timeCalcRtai


.PHONY: clean
```

Debe ser ejecutado con permisos de root y debemos hacerlo así:
make

ANEXO H: RUN

```
#!/bin/sh

libpath=`rtai-config --library-dir`

if [ "$libpath" == "" ]; then
echo "ERROR: please set your PATH variable to <rtai-install>/bin"
exit
fi

export LD_LIBRARY_PATH=$libpath:$LD_LIBRARY_PATH

if [ $UID != 0 ]; then SUDO="sudo "; else SUDO=""; fi
echo
echo "*** TIMECALCRTAI SOFT/HARD PROCESSEs ***"
echo "Press <enter> to load LXRT modules:"
read junk
cd ..
./ldmod
cd -
echo
echo "Now start the realtime process <enter> and wait for its end:"
read junk
$SUDO ./timeCalcRtai
echo
echo "Done. Press <enter> to remove the modules."
read junk
../remod
```

Debe ser ejecutado con permisos de root y debemos hacerlo así: ./run

ANEXO I: Código Prueba Final

```
#!/usr/bin/env python
#Librerias RTAI

from rtai_lxrt import *
from rtai_msg  import *
from rtai_sem import *

#Librerias python
from math import *
from array import *
from thread import *
from random import *

#Librerias PyOPC

from PyOPC.OPCContainers import *
from PyOPC.XDAClient import XDAClient

printf = libc.printf

PERIOD = 64400000 #deadline ns
#PERIOD = 128800000 #deadline ns
ONEMS  = PERIOD
NTASKS = 2
ntasks = NTASKS
LOOPS  = 100

def taskname(x) :
        return (1000 + (x))
```

```
def print_options((ilist,options)):
    print ilist; print options; print

def client_fun(mytask_indx,null2) :

        sem = c_void_p()
        sem2 = c_void_p()
        mytask_name = taskname(mytask_indx)
        mytask = rt_task_init_schmod(mytask_name, 1, 0, 0, 0, 1)

        if mytask == NULL :
                printf("CANNOT INIT TASK %lu\n", mytask_name)
                sys.exit(1)
        printf("THREAD INIT: index = %d, name = %lu, address =
%p.\n", mytask_indx, mytask_name, mytask)
        #Todos los procesos se hacen realtime
        rt_make_hard_real_time()
        #Cuentan los semaforos
        rt_receive(0, byref(sem))
        rt_receive(0, byref(sem2))
        #Todos los hilos una vez creados esperan en este punto
para que corran iguales
        rt_sem_wait(sem2)

        #Crear coneccion
        address="http://192.168.1.2:800"+str(mytask_indx)+"/"

        xda = XDAClient(OPCServerAddress=address,
                                ReturnErrorText=True)
        #print_options(xda.GetStatus())
        #print_options(xda.Browse())

        #hacer periodica la task
```

```
        period = nano2count(PERIOD)
        start_time = rt_get_time() + nano2count(ONEMS)
        rt_task_make_periodic(mytask,start_time,ntasks*period)

        t=0

        count=0
        while count < LOOPS :
                count += 1
                try:


        print_options(xda.Read([ItemContainer(ItemName='sample_i
nteger', MaxAge=500)],
                                                LocaleID='en-us'))



                except Exception:
                        if t<=5:
                                print "ERROR! Intentando conectarse a
Servidor " +str(mytask_indx+1)
                                xda                                    =
XDAClient(OPCServerAddress=address,
                                                ReturnErrorText=True)
                        else:
                                print "No se pudo establecer conexion"
                                rt_task_delete(mytask)
                        t=t+1
                rt_task_wait_period()

        rt_sem_signal(sem)
        rt_make_soft_real_time()
        rt_task_delete(mytask)
```

```
        printf("THREAD %lu ENDS, LOOPS: %d \n", mytask_name,
count)


#main
indx = (c_int*NTASKS)()
mytask_name = nam2num("MASTER")

mytask = rt_task_init_schmod(mytask_name, 1, 0, 0, 0, 0xF)
if mytask == NULL :
        printf("CANNOT INIT TASK %lu\n", mytask_name)

printf("MASTER INIT: name = %lu, address = %p.\n",
mytask_name, mytask)

#semaforos
sem = rt_sem_init(nam2num("SEM"), 0)
sem2 = rt_sem_init(nam2num("SEM2"), 0)

start_rt_timer(0)

for i in range(0, ntasks) :
        indx[i] = i
        start_new_thread(client_fun, (i,NULL))
        while rt_get_adr(taskname(i)) == NULL :
                rt_sleep(nano2count(ONEMS))
        rt_send(rt_get_adr(taskname(i)), c_ulong(sem))
        rt_send(rt_get_adr(taskname(i)), c_ulong(sem2))

rt_sem_broadcast(sem2)

for i in range(0, ntasks) :
        rt_sem_wait(sem)
```

```
        while rt_get_adr(taskname(i)) != NULL :
                rt_sleep(nano2count(ONEMS))

rt_sem_delete(sem)
rt_sem_delete(sem2)
stop_rt_timer()
rt_task_delete(mytask)
printf("MASTER %lu %p ENDS\n", mytask_name, mytask)
```

BIBLIOGRAFÌA

[1] Victor Carceler, “Sistemas Operativos monousuario y multiusuarios”,http://elpuig.xeill.net/Members/vcarceler/c1/didactica/apuntes/ud3/na1

[2] Anónimo,“Kernel Basics, http://commons.wikimedia.org/wiki/File:Kernel_basic.svg

[3] Anónimo, “Kernel basic”, http://commons.wikimedia.org/wiki/File:Kernel_basic.svg

[4] Anónimo,“Planificador”,http://es.wikipedia.org/wiki/Planificador

[5] Abraham Silberschatz, Peter Baer Galvin, y Greg Gagne , “Sistemas de tiempo real”,Prentice Hall, 2005

[6] E. Barrera, “Arquitectura PXI Multiprocesadora para Adquisición y Procesado de Datos en Tiempo Real. Aplicación a Diagnósticos en Entornos de Fusión por Confinamiento Magnético”, Ph. D, Dep. Automa., Univ. Tec. Madrid, Madrid, España, 2008, http://oa.upm.es/1379/1/EDUARDO_BARRERA_LOPEZ_DE_TURISO.pdf

[7] K. Yaghmour, “The Real-Time Application Interface, https://lwn.net/2001/features/OLS/pdf/pdf/rtai.pdf

[8] Anónimo, “RTAI API”, DIAMP, Univ. Pol. Milan, Milan, Italia, http://download.gna.org/rtai/documentation/kilauea/html/sched_overview.html

[9] Anónimo, “Beginner’s Guide”, DIAMP, Univ. Pol. Milan, Milan, Italia, http://www.aero.polimi.it/~rtai/documentation/articles/guide.html

[10] Anónimo, “RTAI API”, DIAMP, Univ. Pol. Milan, Milan, Italia, https://www.rtai.org/documentation/magma/html/api/

[11] E. Bianchi, L. Dozio, D. Martini, P. Mantegazza., “RTAI API”, DIAMP, Univ. Pol. Milan, Milan, Italia, http://www.cs.ru.nl/lab/rtai/

[12] Disponible en: http://cvs.gna.org/cvsweb/?cvsroot=rtai#dirlist

[13] OPC Foundation, “What is the OPC Foundation?”, http://www.opcfoundation.org/Default.aspx/01_about/01_history.asp?MID=AboutOPC

[14] Advosol Inc., “Comparing the XML-DA specification with OPC DA”, http://www.advosol.us/driver.aspx?Topic=CompareXMLDA

[15] OpenOPC project., “OpenOPC Command-line Client”, http://openopc.sourceforge.net/client.html

[16] Hermann Himmelbauer,“PyOPC. A Python Framework for the OPC XML-DA 1.0 Standard”, 2006, http://pyopc.sourceforge.net/docs/html/index.html