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

Lomas Ascencio Henry Antonio (1), Ochoa Cevallos Denisse Adrianna (2) , Ochoa Donoso Daniel (3)
Facultad de Ingeniería en electricidad y computación(FIEC)
Escuela Superior Politécnica del Litoral (ESPOL)
Campus Gustavo Galindo, Km 30.5 vía Perimetral
Apartado 09-01-5863. Guayaquil-Ecuador
heanloma@espol.edu.ec (1), aochoa@espol.edu.ec (2), dochoa@espol.edu.ec (3)

Resumen

Se realizó un sistema de control simple conectado en red siendo monitoreado por un programa informático desarrollado bajo 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. Se realiza una serie de experimentos comparativos de la combinación de lenguajes de programación como C y Python en conjunto de RTAI así como los protocolos de comunicación OPC DA y XML. Al final presentamos un sistema de control óptimo con las herramientas seleccionadas y las conclusiones del mismo.

Palabras Claves: *Hard Realtime, Soft Realtime, RTAI, Kernel, Planificador, Procesos, C-RTAI, Python-RTAI, OPC, XML.*

Abstract

We realized a simple control system networked being monitored by a software tool developed under RTAI. It combines the use of programming techniques and control system simulated by software. The scheme is the basic client-server relationship, server simulates an OPC, which abstracts us from connecting to a PLC directly, for example, client and our software tool is developed to read data from OPC in real time. A series of comparative experiments combining programming languages like C and Python under RTAI and communication protocols like OPC DA and XML is performed. Finally we present an optimal control system with the selected tools and conclusions.

Keywords*: Hard Realtime, Soft Realtime, RTAI, Kernel, Scheduler, Process, C-RTAI, Python-RTAI, OPC, XML.*

1. Introducción

Los sistemas informáticos que necesitan un alto grado de precisión para sus aplicaciones en las industrias o campos médicos dependen no solo en la efectividad y veracidad de sus datos, sino también de la prontitud con la que el sistema responde, es decir, la responsabilidad recae en entregar los datos antes de que se cumplan un tiempo máximo de respuesta; luego de ese límite de tiempo por muy acertados que sean lo datos estos ya no son útiles.

Los Sistemas en Tiempo Real garantizan entregar resultados correctos y dentro del dicho límite de tiempo valiendose de varios métodos e inclusive de una interfaz como el caso de RTAI (RealTime Application Interface) y de técnicas de programación como la planificación de procesos que permitan darles mayor prioridad de ejecución sobre otros no tan relevantes para el propósito..

2. Información General.

Se utilizaron herramientas como RTAI para el sistema en tiempo real haciendo uso de técnicas sobre planificación de procesos, sistema OPC simulado para emular un PLC que constantemente este proporcionando datos tanto en protocolo DCOM con el Matrikon así como XML con OpenOPC.

Se busca 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, estudiar los Sistemas OPC y sus protocolos de comunicación que permita crear un sistema simulado OPC con el que pueda comunicar satisfactoriamente un cliente OPC y un servidor OPC.

El sistema OPC es simulado en software así como los dispositivos conectados en red con tráfico de red y red dedicada.

3. Marco teórico

Sistema Operativo. 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]

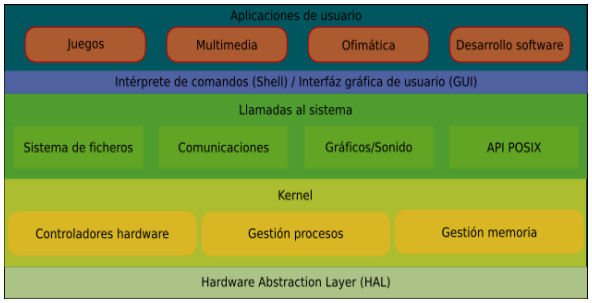

Fig. 1 Componentes de un Sistema Operativo [1]

Kernel. El kernel o núcleo es la parte fundamental de un sistema operativo. Se encarga de la comunicación entre las aplicaciones y el hardware. 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 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 [2].

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. Las políticas de planificación pueden ser: FIFO (First In, First Out), Round Robin, planificación por prioridades, entre otras.

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. [3]

Sistema en tiempo real. 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.

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 (hard real-time) 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 (soft real-time) 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.

Para la implementación de un sistema en tiempo real se debe enfocar en los procesos y en sus prioridades de ejecución en el kernel, minimizar retardos, minimizar interrupciones de periféricos, entre otros. Las características para cumplir dicho acometido son:

- Apropiación de kernel
- Planificación de procesos
- Latencia minimizada

Apropiación del kernel. La Apropiación del kernel consiste en desalojar una tarea en ejecución en el kernel si un proceso de mayor prioridad necesita ser ejecutado. En caso de no tener un kernel apropiativo la tarea debe esperar a que la tarea que se encuentra activa en el kernel termine su ejecución y dé paso a la siguiente así esta sea de mayor prioridad. Una estrategia para lograr que un kernel sea apropiativo 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.

Planificación de Procesos. La planificación de procesos se vale de algoritmos que permite darle a los procesos más importantes la prioridad más alta sobre los procesos menos importantes. De esta forma sólo se garantiza la funcionalidad de los sistemas en tiempo real no estrictos.

Latencia. La latencia consiste en el tiempo transcurrido entre que se da un suceso y en que este es atendido.

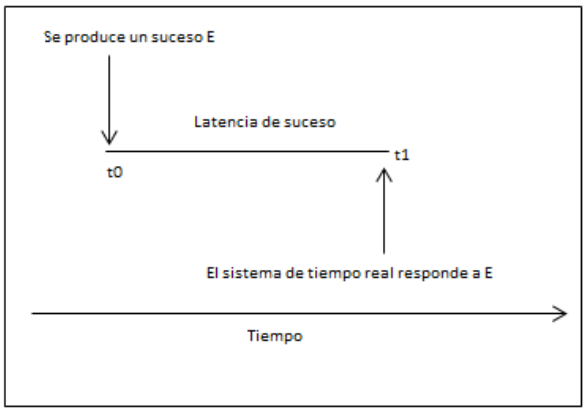

Fig. 2 Latencia de suceso

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. 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.

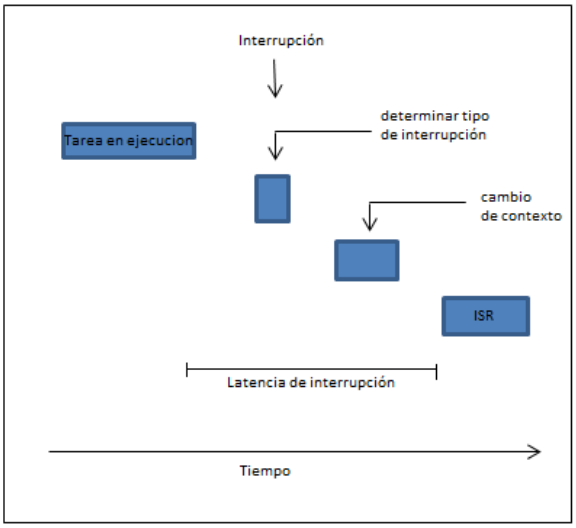

Fig. 3 Latencia de interrupción

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.

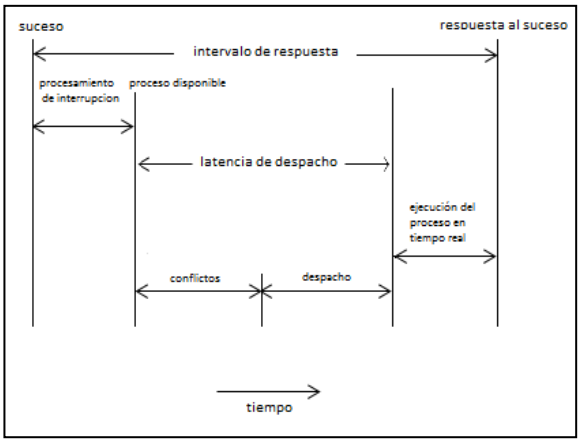

Fig. 4 Latencia de despacho

Para contar con un sistema en tiempo real, uno de los objetivos en minimizar la latencia para atender a procesos de alta prioridad tan pronto sea posible.

Planificación de la CPU. Para la planificación de la CPU se deben considerar 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.

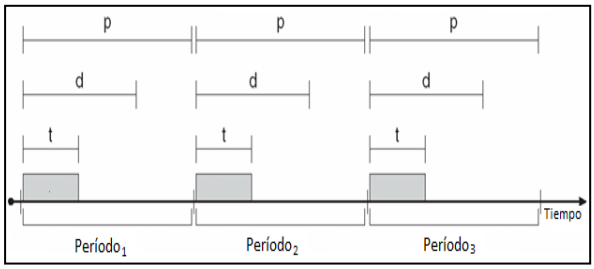

Fig. 5 Planificación de la CPU en Tiempo Real

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.

Es necesario realizar una planificación de procesos para que estos tomen tiempo de ejecución cuando sean necesarios y maximizar el uso de la CPU.

Planificación por prioridad monótona en tasa. Consiste en planificar tareas periódicas. Proporciona 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.

4. 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).

ArquitecturaRTAI 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.

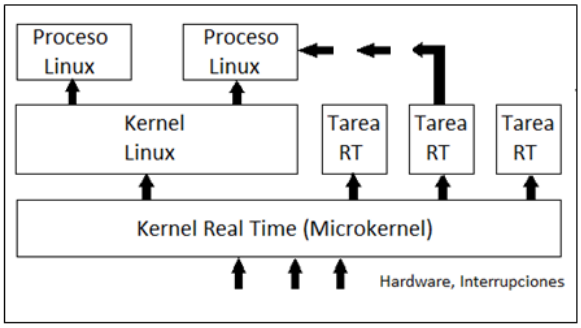

Fig. 6 Funcionamiento de RTAI

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) [4].

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

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.

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.

5. Open Plataform Communications

OPC, por sus siglas en inglés, es una especificación de un protocolo de comunicaciones de tiempo real que crea un interfaz 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 también plataformas de Microsoft que 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.

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, la arquitectura básica sería muy parecida a esto.

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.

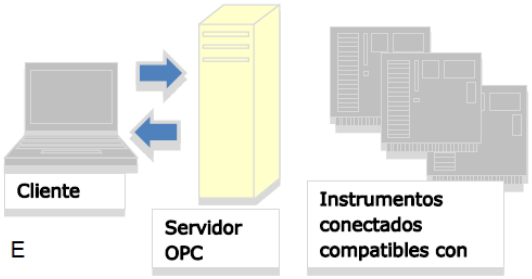

Figura 7 Arquitectura básica de un sistema OPC

Advosol [7] 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 [8] [9] [10].

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

Tabla 8. Creación de una instancia del Servidor almacenada en un objeto

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

Tabla 9. Conectarse a un 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'))

Tabla 10. Leer datos de ese servidor.

Soporte de lenguajes de programación en RTAI. 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, 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 nuestros experimentos pudimos analizar cómo se desempeñan estas dos herramientas realizando pruebas comparativas, bajo ciertas condiciones que demuestren sus ventajas en nuestros estudios.

6. Experimentos

Para analizar cómo se desempeñan estas dos herramientas se realizó pruebas comparativas, bajo ciertas condiciones que demuestren sus ventajas en nuestros estudios..

6.1. Justificación del uso de RTAI

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.

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. Tabla [11]

6.2. Comparar rapidez de respuesta entre Python-RTAI Hard Realtime y Python-RTAI Soft Realtime

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. Tabla [12]

6.3. 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. Tabla [13]

6.4. Comparar rapidez de lectura entre los protocolos XML-DA y OPC DA

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. Tabla [14]

6.5. 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.

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.

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.

en la cual necesitamos un período p, un tiempo de lectura t, un tiempo límite d obtenidos en los experimentos #3 y #4 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. Figura 8

9. Resultados

	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 11. Tiempos promedios API Estándar vs API RTAI en Soft y Hard realtime.

	HARD REALTIME	SOFT REALTIME
PROMEDIOS	190 ns	260 ns
σ (MÁXIMO)	6.5491E-08	4.0482E-04

Tabla 12.Tiempo promedio Python-Rtai en Soft realtime versus Hard 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 13. Comparación C-RTAI versus Python-RTAI.

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

Tabla 14. Tiempos promedio de lectura PyOCP versus OpenOPC.

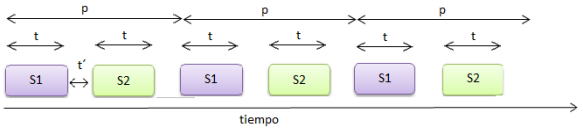

Figura 8 Esquema de planificación de la comunicación de los servidores

10. Recomendaciones

Tener servidores en diferentes PCs y en el cliente tener varias tarjetas de red, una por servidor.

Realizar lecturas con servidores de diferentes tipos de protocolo, es decir, uno PyOPC y otro OpenOPC al mismo tiempo.

Buscar una solución que permita hacer uso del lenguaje C.

Tener un hilo que permita controlar el tiempo máximo que debería tomar una lectura en un servidor.

11. Conclusiones

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

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.

El servidor implementado bajo el protocolo XML-DA responde en la mitad del tiempo que el implementado con protocolo OPC-DA.

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.

Aún con tráfico de red un servidor implementado con XML-DA es 40% más rápido en responder que uno implementado con OPC DA.

Un servidor implementado con OPC DA mantiene su constancia de respuesta frente al tráfico de red.

Un servidor XML-DA es susceptible al tráfico de red, pierde constancia en su tiempo de respuesta.

Incluso programando con Python un sistema hard realtime tiene la ventaja de ser estable en sus resultados volviéndolo predecible y controlable.

La rapidez depende del desempeño del lenguaje empleado, sea Python-RTAI o C-RTAI.

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.

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.

El lenguaje C aún con herramientas de RTAI se mantiene superior en rapidez de respuesta a Python con herramientas RTAI.

12. Referencias

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

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

[3] Anónimo, "Planificador",http://es.wikipedia.org/wiki/Planificador

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

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

[8] OpenOPC project., "OpenOPC Command-line Client", http://openopc.sourceforge.net/client.html

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