RACK: Plataforma de desarrollo rápido de aplicaciones en tiempo real

Jorge Nieto[1*] Sebastian Smolorz[1] Bernardo Wagner[1]
(1) Grupo de Investigación en Sistemas en Tiempo Real RTS
Leibniz Universität Hannover, Appelstraße 9ª Alemania
*nieto@rts.uni-hannover.de

Recibido 17 de Octubre de 2010. Aceptado 01 de Mayo de 2011
Received: October 17, 2010 Accepted: May 01, 2011

RESUMEN

Con los constantes incrementos en la complejidad de los sistemas automatización y sus acelerados ciclos de desarrollo se incrementa también la importancia de facilitar y optimizar el software respectivo. En este artículo se presenta al *framework* RACK (*Robotics Application Construction Kit*), un marco de construcción desarrollado para aplicaciones de robótica en tiempo real que puede ser aplicado en varias áreas del control automático. El sistema soporta el desarrollo de componentes tanto de tiempo real estricto como de tiempo no real y define todos los elementos de programación necesarios (drivers, módulos para realizar funciones de navegación y herramientas para programar interfaces gráficas de usuario) para implementar sistemas complejos. Además, el RACK provee un servicio de comunicación en tiempo real entre componentes. El presente artículo explica el diseño, así como la implementación de RACK y describe un robot autónomo de ejemplo que usa este sistema.

Palabras clave: Robots autónomos de servicio, arquitectura de tiempo real, sistema interconectado multi-robot, Xenomai.

RACK: Framework for construction of real-time applications

ABSTRACT

With the constant increasing in the complexity of automation systems and their accelerated development cycles, the need for optimizing and making the respective software more accessible increases. This article presents the Robotic Applications Construction Kit (RACK). The framework supports the development of real-time and non-real-time components and defines all necessary programming elements to implement complex systems. Besides, RACK provides a real-time, inter-component messaging service. The present article describes the design and implementation of RACK and describes an example of an autonomous mobile robot based on this framework.

Keywords: Autonomous service robots, real-time architecture, interconnected muti-robot system, Xenomai.

1. INTRODUCCIÓN

Una plataforma robótica móvil es un sistema complejo compuesto esencialmente por diferentes sensores, un conjunto de actuadores y un sistema de control que supervisa las entradas y genera acciones de control para guiar el movimiento del robot de acuerdo a una tarea predeterminada. Los robots móviles autónomos encuentran aplicación en diversos escenarios, por ejemplo como guía de visitas a museos [1], ayudante en la realización de tareas domésticas, o incluso como medio de optimización de procesos logísticos industriales, como lo es el caso del montacargas autónomo desarrollado como producto de la cooperación entre el grupo de investigación en sistemas de tiempo real (RTS) de la Universidad de Hanóver y la empresa Still GmbH de Alemania, y que se encuentra operando en el área de producción [17].

Para lograr el control de tales robots se requieren sistemas de cómputo que realicen funciones como fusión sensorial, percepción y navegación. Además se requiere un procesamiento determinístico implementado en controladores distribuidos, lo que implica la necesidad de comunicación en tiempo real.

El grupo de investigación RTS desarrolló el Kit de Construcción de Aplicaciones para Robótica RACK (pos sus siglas en inglés *Robotics Application Construction Kit*), que es un *framework* compacto diseñado para cumplir dichos requerimientos y que tiene el fin de facilitar la implementación de sistemas de software para robots autónomos y su investigación. Con la ayuda de RACK, se pueden implementar sistemas de control tanto en plataformas centralizadas (un solo sistema de cómputo) como en distribuidas. RACK provee las funciones elementales necesarias para monitorear y controlar robots autónomos en tiempo real.

Actualmente existen varias implementaciones similares diseñadas para ciertos tipos de robots en particular. El sistema operativo de código abierto para aplicaciones robóticas, OROCOS (por sus siglas en inglés) [8], por ejemplo, fue desarrollado para satisfacer los requerimientos de una amplia gama de software de robótica y control. Dicho *framework* soporta funcionalidades y servicios relativos a la ingeniería de control. Sin embargo, este carece de las interfaces y los componentes adecuados para permitir su uso en robots móviles.

La plataforma RT-CORBA [6] es una extensión de tiempo real para el *middleware* CORBA [9]; no obstante, esta plataforma no estaba disponible para Xenomai [13] (la extensión de tiempo real para Linux), que es el sistema usado en los robots del RTS . Además de esto, los autores consideran que RT-CORBA es demasiado extenso para los escenarios de aplicación estudiados en el instituto y, por lo tanto, difícil de aprender para los principiantes.

Existe además un buen número de proyectos de software para robótica disponibles, como por ejemplo el "*Dave's Robotic Operating System*" [10], el proyecto "*Player/Stage*" [11], o el "*Mobile and Autonomous Robotics Integration Environment* (MARIE) [12]. No obstante, ninguno de ellos cumplía con los requerimientos del procesamiento distribuido en tiempo real estricto.

En la siguiente sección se describe el diseño del *framework* genérico RACK. La sección III da una visión general de las abstracciones disponibles de *hardware* y de las estructuras de datos específicas para robots móviles, así como de sus interfaces. Aspectos referentes a la implementación de componentes así como los requerimientos de RACK y la implementación interna se presentan en la sección IV. Un sistema robótico de ejemplo basado en RACK se describe en la sección V. El presente artículo concluye con un resumen y una perspectiva de las futuras mejoras al sistema presentado.

2. DESCRIPCIÓN DEL FRAMEWORK

En esta sección se describe la arquitectura del sistema RACK. Dicha arquitectura está construida de forma modular con un sistema de comunicación entre componentes basado en mensajes. Cada componente representa una unidad de procesamiento de datos que realiza una tarea específica, como el manejo de un dispositivo

(*driver*), el registro de datos, control automático y despligue de la información (interfaz gráfica), entre otros. Por lo tanto, algunos de los componentes de RACK son considerados de alto nivel y otros como de bajo nivel, estableciéndose varias capas, como se explica en la siguiente sección. La comunicación se realiza a través de un servicio de mensajería con capacidades de tiempo real, el cual fue diseñado en conjunto con la plataforma.

Cada sistema de cómputo basado en RACK puede funcionar ya sea de manera autónoma como un sistema robótico, o como unidad de procesamiento de datos (subsistema) para un sistema robótico en un nivel superior. Por ejemplo se puede tener un computador embebido implementando una arquitectura RACK, el cual tiene la única tarea de adquirir y procesar datos de un sensor láser 3D. Este PC puede a su vez conformar un sistema de adquisición para un sistema robótico con más procesadores distribuidos, el cual toma y recibe información y la emplea para realizar tareas de navegación. El sistema RACK además provee de una interfaz de usuario, igualmente modular, que permite la búsqueda de errores y el monitoreo del sistema.

2.1 . Arquitectura basada en componentes

Un sistema robótico convencional está conformado por varios subsistemas, ya sea de *hardware* o de *software*, como por ejemplo servomotores, sensores, procesamiento de los datos de entrada, controladores, comunicación con el usuario, entre otros. El sistema RACK permite una abstracción de cada uno de estos elementos, que en el contexto del *framework* se denominan *componentes*.

Cada componente puede ser una unidad autónoma operando en su propia unidad de cómputo huésped (a partir de ahora se usará el término *host*) y a su vez diferentes *hosts* pueden estar integrados en un robot móvil. Incluso varias plataformas robóticas puede conformar un grupo de robots cooperativos.

La arquitectura RACK establece una encapsulación para los componentes, donde cada uno de ellos no puede acceder directamente a la memoria de los otros.

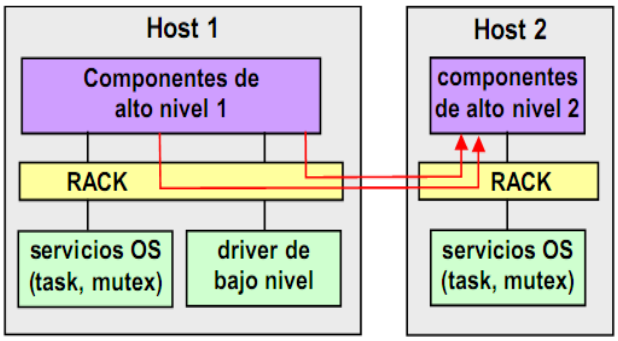

Fig. 1. Diagrama esquemático de la arquitectura RACK para un sistema de dos *hosts*.

Un sistema robótico además se puede considerar desde el punto de vista jerárquico conformado principalmente por un nivel alto, donde ocurre la supervisión, uno nivel medio, donde ocurre el control, uno medio-bajo, correspondiente a los servicios del sistema operativo y un bajo nivel, donde se encuentran los *drivers* cercanos al *hardware*.

RACK se trata de un *middleware*, esto es, una capa de mediación entre los niveles medio-bajo y superior mencionados arriba, como se muestra en la Fig. 1. El framework ofrece no solo servicios de tiempo real, como el manejo de tareas y los mecanismos de comunicación y sincronización (mutexes, semáforos), sino también interfaces bien definidas para manejo de dispositivos, como por ejemplo bus CAN y puertos serie, entre otros. La Fig. 2 muestra un ejemplo simplificado de un robot distribuido basado en RACK. La comunicación entre componentes ocurre exclusivamente a través de un servicio de mensajería análogo al sistema de reparto de correo. Este servicio, denominado TiMS se explica más adelante.

2.2 . Comunicación entre componentes

RACK ofrece un mecanismo consistente para implementar la comunicación entre componentes simplemente usando buzones de correo (*mailboxes*). Cada componente usa un buzón primario para recibir comandos encapsulados en forma de

mensajes, los cuales han sido enviados por otros módulos (ver numeral 1). Cada vez que se requiera, se pueden crear nuevos buzones para permitir la recepción de datos de los otros componentes o para satisfacer los requerimientos individuales de cada componente.

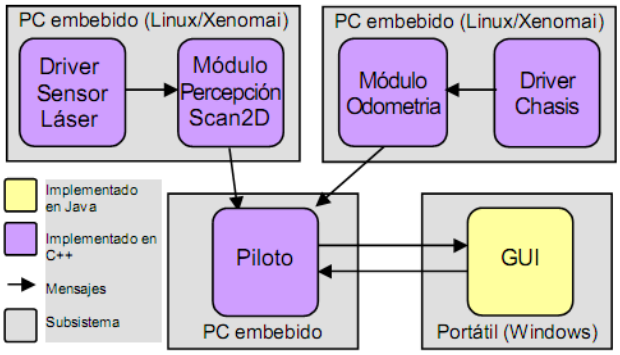

Fig. 2. Diagrama esquemático simplificado de un sistema distribuido usando RACK.

El *framework* define además estructuras de datos correspondientes a los mensajes. Estos tipos son útiles en el envío de comandos y de respuestas, de acuerdo al esquema de llamado a procedimiento remoto (RPC) [21], así como de mensajes en una sola dirección.

El intercambio de datos se puede realizar a través de un modelo de consulta (*polling*), en donde el componente que requiere datos envía una solicitud a otro y espera a que este le envíe un mensaje de datos como respuesta. Por otro lado, es posible también el intercambio de datos a través de un modelo *publicar/suscribir* (*publish/subscribe*) [22], en donde uno o más componentes (*suscriptores*) pueden leer de un flujo de datos (*stream*) generado por otro componente (*publicador*). Por convención un componente no puede realizar el envío de información sin que un modulo receptor se lo haya pedido explícitamente.

```
struct message_header {
  __u8  flags;    // flags del mensaje
  __s8  type;     // tipo del mensaje
  __s8  priority; // prioridad del mensaje
  __u8  seq_nr;   // número de sequencia
  __u32 dest;     // dirección de destino
  __u32 src;      // dirección de orígen
  __u32 msglen;   // longitud
};
```

Fig. 3. Cabecera de un mensaje en RACK.

Debido al carácter de sistema distribuido, los componentes pueden estar repartidos en diferentes arquitecturas de computadores con diferentes formatos de orden de *byte*, por lo cual, los mensajes recibidos por un componente deben ser convertidos al formato de orden byte local al momento de su recepción, en caso de que el emisor del mensaje use un formato diferente.

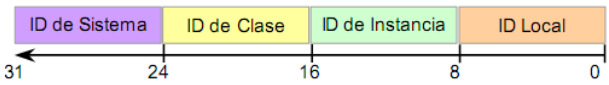

Fig. 4. Estructura de una dirección en RACK.

1) Mensajes: A cada mensaje se le da un formato consistente, el cual consta de un encabezado de 16 bytes y una secuencia de datos de tamaño variable. El encabezado contiene información de enrutado (dirección de origen y destino), la longitud del mensaje, su tipo, la prioridad en el canal de transmisión y el *mailbox* destino, un número de secuencia y bits de marca (*flag bits*), los cuales codifican el orden de *byte* del mensaje (Fig. 3). La dirección de un *mailbox* es creada de acuerdo al nombre de la clase correspondiente, la instancia en ejecución y el sistema donde se está ejecutando. Esto se describe en el siguiente ítem. RACK usa internamente un cierto rango de direcciones para implementar servicios de núcleo (ver 2.6-1). Por lo tanto, se provee un *offset* especial para tipos definidos por el usuario.

En cuanto al cuerpo del mensaje, este puede contener información o un comando; los comandos de salida, es decir aquellos que son enviados a otros componentes son codificados como tipos positivos mientras que los mensajes de entrada o respuesta se codifican como negativos.

2) Nomenclatura de las direcciones: Para poder acceder de forma global a los componentes, se requiere de identificadores. Estos identificadores corresponden a las direcciones principales de *mailbox* de los componentes.

Una dirección de *mailbox* o un identificador de componente consiste de cuatro campos de identificación o IDs, a saber: el ID de sistema, el ID de instancia, el ID de clase y el ID local. Cada uno de estos tiene un *byte* de longitud. Sin embargo, al

diseñar un nuevo sistema robótico basado en RACK, se puede fijar individualmente el tamaño de los campos. La Fig. 4 muestra la segmentación de las direcciones de buzón usadas por los sistemas robóticos en el RTS.

Por medio del ID de sistema, se pueden distinguir inequívocamente diferentes sistemas robóticos o de cómputo. El ID de clase define la clase componente genérica, como se explica posteriormente en la sección 3.1. Para usar varios componentes de la misma clase en un sistema robótico, a cada componente se le asigna un ID para representar la instancia correspondiente. El ID local se usa para direccionar los distintos *mailboxes* en un componente, empezando por 0 para el buzón de recepción de comandos.

3) Enrutado: La distribución de mensajes hacia los sistemas remotos se realiza a través de un enrutado jerárquico. Cada plataforma de computador basada en RACK está provista de un enrutador de mensajes (*router*) que registra los buzones de todos sus componentes locales. Para enviar mensajes a otras plataformas o sistemas, se debe conocer la dirección del enrutador correspondiente en el siguiente nivel. Cada enrutador informa al enrutador de más alto nivel, acerca de la presencia de todos los buzones de componentes reconocidos. Por lo tanto el enrutador de más alto nivel de un sistema robótico se encarga de registrar a todos los buzones disponibles de otros sistemas robóticos conectados a este. Se puede realizar un enrutado sistema-a-sistema usando explícitamente la ID de sistema. En la Fig. 5, se muestra un ejemplo de enrutado jerárquico.

En el *framework* RACK existen los siguientes tipos de enrutadores:

Enrutador Básico – Un subsistema que envía mensajes con una dirección de destino desconocida a un enrutador de alto nivel dado.

Enrutador de Sistema – Enrutador de más alto nivel de un sistema robótico individual. Todas las direcciones de buzón correspondientes al sistema local son registradas por este enrutador. Un enrutador de sistema puede recibir y reenviar mensajes de los enrutadores de un subsistema relacionado así como de otros sistemas externos.

Enrutador de Alto Nivel – Este enrutador registra los *mailboxes* de todos los sistemas robóticos usados. Si el enrutador de alto nivel no conoce la dirección de destino, el mensaje es descartado. El *router* de más alto nivel del sistema completo se denomina *Enrutador Superior*.

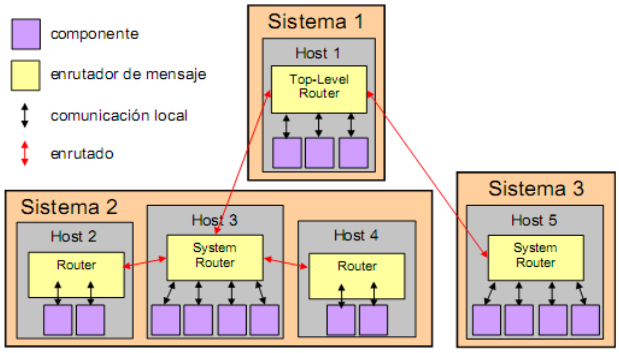

Fig. 5. Flujo de datos en un enrutado jerarquico.

2.3 . Modelo de ejecución

Hasta el momento se han considerado los componentes de forma abstracta. En esta sección se entra en detalle en cuanto a la ejecución del *software* del sistema. Desde el punto de vista del manejo de mensajes, se tienen dos tareas principales, la tarea de comandos (*commandTask*) y la tarea de datos (*dataTask*). Un componente genérico ejecuta *commandTask*, la cual bloquea indefinidamente hasta que recibe un mensaje, después de lo cual ejecuta las rutinas de manejo o comandos *proxy* (*proxy handlers*). Una vez la rutina es finalizada, la tarea vuelve a bloquear esperando al siguiente comando.

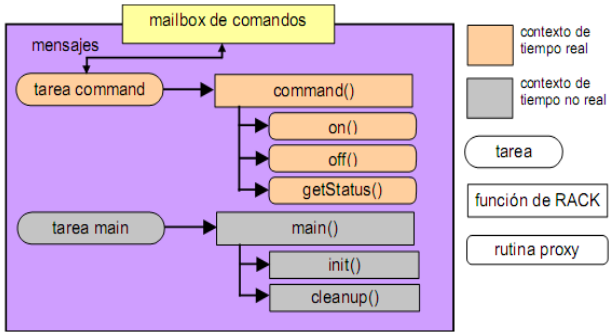

Fig. 6. Diagrama funcional de un módulo en RACK.

Por otro lado, los componentes encargados de manipular y generar datos, ejecutan la tarea adicional *dataTask*, que les permite solicitar y recibir datos de otros componentes. Estos datos son entonces procesados por el componente y almacenados en un *buffer* específico.

Dentro de los componentes se pueden implementar otras tareas con funciones específicas (por ejemplo un *watchdog*). Es posible además ejecutar tareas con una prioridad más alta. A diferencia del *framework* de *software* que fue usado anteriormente en el RTS [2], la prioridad de las tareas no está fijada en el código por medio de la ID de clase. De esta manera, se pueden implementar flujos de datos (*data streams*) incluyendo diferentes componentes por medio de una asignación de la prioridad.

2.4 . Interfaces de Módulos Genéricas

Un componente RACK provee dos interfaces de aplicación diferentes. Una de ellas (*Interface RACK*) es usada directamente por el *middleware* para el arranque, inicio y liberado de recursos (*clean up*) del componente, mientras que la segunda (*Interface Proxy*) se requiere para dar al componente la posibilidad de comunicarse con otros por medio de funciones *proxy*.

La Fig. 6 muestra todas las tareas y funciones de un módulo genérico.

1) Interface RACK: el *framework* usa las siguientes funciones dentro de un módulo genérico:

init() – Durante la carga de un componente, la función *main()* del componente ejecuta esta función con el fin de reservar memoria para todas las estructuras de datos, incializar dispositivos y para realizar otras tareas de inicialización las cuales no necesariamente deben ocurrir en el contexto de tiempo real.

command() – Esta función es llamada por una tarea cada vez que un nuevo mensaje RPC es recibido. Además la función ejecuta rutinas de manejo *proxy* de acuerdo a los comandos del módulo.

cleanup() – Durante la finalización o descarga de un componente en memoria, la función *cleanup()* es llamada por la rutina de manejo de señal con el fin de "borrar" el módulo, es decir, finalizar tareas iniciadas por este y reinicializar variables internas.

Solamente la función *main()* tiene que ser implementada al crear un nuevo componente. Todas las otras funciones genéricas son parte integral de RACK, sin embargo pueden ser redefinidas para realizar funciones específicas.

2) Interface Proxy: La tarea *commandTask* ejecuta las siguientes rutinas de manejo de acuerdo a los comandos de *proxy* recibidos:

on() – Si un componente es activado por otro, esta rutina prepara el módulo para que pueda ser ejecutado en el contexto de tiempo real.

off() – Un componente puede ser desactivado o "apagado" por medio de esta rutina. La tarea *comandTask* ejecuta *off()* para deshacer todos los pasos previamente realizados en la función *on()*.

getStatus() – Esta rutina *proxy* devuelve el estado actual del módulo: *ENABLED*, si el módulo está habilitado, *DISABLED*, si está deshabilitado o *ERROR*, en caso que un error haya ocurrido y el módulo no esté operando en modo normal.

Estas rutinas genéricas de *proxy* son también parte integral de RACK. Las funciones *on()* y *off()* se pueden sobre-escribir por ejemplo para solicitar continuamente datos de otros componentes. Las rutinas de manejo *proxy* son llamadas por la tarea *commandTask*, por lo tanto estas **no deben** bloquear de manera indefinida.

2.5 . Interfaces de Módulos de Datos

Los módulos almacenan datos así como las correspondientes marcas de tiempo en su *buffer* de datos. El número de conjuntos de datos y por consecuente el historial de marcas de tiempo depende del tamaño de *buffer* empleado. Otros componentes pueden solicitar tanto datos correspondientes a una marca de tiempo dada (*polling*) así como todo un flujo continuo de datos. Las marcas

de tiempo se necesitan por ejemplo para fusión sensorial, facilitando el sincronizado de los datos.

Al contrario del módulo genérico, un módulo de datos tiene funcionales de *interface* adicionales, las cuales se describen en esta sección. La Fig. 7 muestra todas las tareas y funciones de un módulo de datos.

1) Interface RACK: La siguiente función se debe implementar dentro de un nuevo módulo de datos, adicional a la *interface* de módulo genérica:

loop() – En un ciclo periódico negociado, la tarea *data task* ejecuta esta función para obtener los datos necesarios provenientes de *drivers* u otros componentes. La tarea *data task* tiene que estar corriendo para que los datos creados o manipulados puedan ser almacenados en un *buffer* de datos específico.

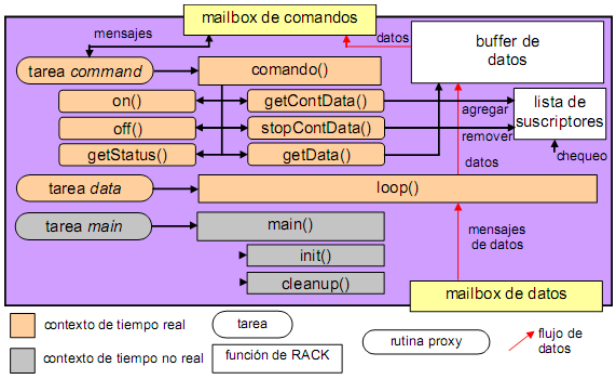

Fig. 7. Diagrama funcional de un módulo de datos en RACK.

2) Interface Proxy: Un componente que produce o manipula datos tiene que proveer rutinas de manejo tipo *proxy* para implementar el modelo *polling* así como el de *publica/suscribe*. Estas rutinas son parte de RACK y no pueden ser sobrescritas:

getData() – Para facilitar la implementación de un modelo *polling* se provee esta rutina. Otros componentes pueden solicitar datos correspondientes a una marca de tiempo dada. La tarea *dataTask* busca dentro del *buffer* de datos la información adecuada y la envía al receptor en espera (modo síncrono).

getContData() – Usando *getContData()*, los componentes receptores solicitan un flujo continuo de datos. Esta rutina agrega el recipiente a una lista de "subscriptores" (*listeners*), después de esto se lleva a cabo una negociación del tiempo de período por medio de la recepción del período deseado y el envío de un valor de tiempo sugerido soportado por este componente. Si hay datos disponibles y se satisface la condición de tiempo, *dataTask* envía los datos generados al recipiente (modo asíncrono).

stopContData() – Esta función es llamada por el componente receptor para cerrar un flujo de datos, con lo cual este receptor es removido de la lista de suscriptores.

En la Tabla 1 se listan todas las rutinas genéricas y tipo *proxy*.

Tabla 1. Rutinas genéricas de un componente RACK.

Función	Módulo
on()	genérico
off()	genérico
getStatus()	genérico
getData()	datos
getContData()	datos
stopContData()	datos

2.6 . Servicios Adicionales

1) Servicio de registro de datos (*data logging*): Un problema práctico en el desarrollo de sistemas distribuidos es el seguimiento de toda la información y mensajes de depurado (*debugging*) desplegados en diferentes pantallas o mostrados en consolas remotas por diferentes componentes. Por este motivo, se requiere de un servicio seguro de registro en tiempo real para colectar todos los mensajes de información y depurado y mostrarlos en una *interface* central de usuario. La cantidad de información enviada por este servicio debe ser fijada usando niveles de información. Los componentes correspondientes pueden ser así analizados, además se pueden detectar errores de manera más simple para ser corregidos sistemáticamente. Toda la información puede ser

grabada en un dispositivo de almacenamiento para luego ser analizada.

2) Detección de Cambio de Contexto: Es importante asegurase de que todos los componentes están siendo ejecutados en un estado predefinido. Tareas de tiempo crítico deben correr continuamente en el contexto de tiempo real. Un cambio indeseable de contexto debe ser detectado y notificado. RACK es capaz de detectar estos cambios de contexto y forzar una búsqueda inversa (*back-tracing*) para hallar y corregir tales errores no obvios. La detección también resulta de ayuda para validar librerías desconocidas relacionadas con las capacidades de tiempo real del sistema.

3) Búsqueda Inversa: Después de la aparición de eventos inesperados o de errores, se inicia un servicio de *back-tracing* para asistir al programador en la búsqueda de la causa del error de manera más rápida. RACK muestra en la consola la pila de las últimas funciones que fueron llamadas a fin de poder analizar errores como fallos de segmentación y cambios de contexto.

4) ***Interface* de Argumentos:** RACK provee una *interface* para transferir argumentos específicos y parámetros a los componentes. Para este propósito, se da una estructura de datos para definir estos argumentos y guardar sus valores. Mientras se inicia un componente, el *middleware* analiza todos los argumentos y parámetros dados y escribe los valores correspondientes en la estructura de datos. Los componentes pueden acceder a estos valores directamente.

2.7 . Interface gráfica de usuario

La *interface* gráfica de usuario (GUI) soporta el desarrollo de sistemas robóticos proveyendo las siguientes funcionalidades:

- Análisis de diagnóstico de los componentes
- Configuración de los componentes
- Presentación visual de la información del sistema.

El otro propósito de la GUI es la operación de un sistema robótico autónomo complejo de manera fácil para el usuario, por medio de las siguientes funcionalidades:

- Operación de diversos sistemas robóticos.
- Entrada de puntos de ruta para navegación
- Control de un chasis por medio de un *joystick* virtual

3 . COMPONENTES DEL SISTEMA RACK

Un componente RACK es la unidad ejecutable más pequeña de un sistema robótico implementado con RACK. Debido a la encapsulación, los componentes pueden ser sustituidos por otros sin necesidad de cambiar el sistema completo. Además, es posible implementar los componentes en diferentes lenguajes de programación y correr estos módulos en sistemas distribuidos.

En esta sección se describe la división de estos componentes en clases y la implementación de la comunicación entre ellos por medio de *proxys.*

3.1 . Clases RACK

Las clases de componentes pueden representar clases abstractas de *hardware* como sensores y actuadores así como módulos de procesamiento de datos. Como ejemplo, componentes tipo *driver* para sensores láser como los fabricados por SICK [18] o Ibeo [19] son asignados a la clase *Ladar*. La comunicación entre componentes de diferentes clases ocurre exclusivamente a través de las *proxys* descritas en la siguiente sección.

En la Tabla 2 se muestran algunas de las clases actualmente definidas.

3.2 . Proxys RACK

Definiciones específicas de clase como los son los tipo mensaje, estructuras de datos (incluyendo sus

funciones de análisis sintáctico o *parsing*) y funciones de *proxy* soportadas son definidas en un *proxy* específico a cada componente RACK adicional a las *proxys* genérica y de datos descritas en la sección anterior. Debido a las clases abstractas, solo se requiere un *proxy* para acceder a todos los componentes similares dentro de una clase. Algunas clases tienen que extender la API *proxy* genérica, por ejemplo la clase *Chassis*.

Tabla 2. Rutinas genéricas de un componente RACK

Clase	Tipo
Camera	Driver
Chassis	Driver
GPS	Driver
Gyro	Driver
Joystick	Driver
Ladar	Driver
Odometry	Driver
ServoDrive	Driver
Path	Navigation
Pilot	Navigation
Position	Navigation
Scan2D	Navigation
Scan3D	Perception
ObjectRecognition	Perception

Para poder enviar comandos al componente *Chassis* y cumplir con tareas especiales de navegación se debe fijar de antemano un piloto por medio de la GUI. Por lo tanto, se provee la función *setActivePilot*(), perteneciente al *chassisProxy*, así como la estructura y la función de análisis sintáctico correspondientes. Cuando se usan las funciones *proxy*, los datos recibidos son convertidos automáticamente al tipo local.

4. IMPLEMENTACIÓN DE SOFTWARE

El RTS trabaja con PC industriales embebidos denominados *Cajas de Procesamiento Escalables* o *Scalable Processing Boxes,* abreviado SPB (Fig. 8) [7]. Estos están equipados con procesadores x86 y tarjetas de expansión PC/104 y además incluyen puertos CAN y tarjetas de expansión para puerto serie rápido. La velocidad de los procesadores de núcleo simple y doble varía de 266MHz a 2.2 GHz. El sistema operativo y el *software* de aplicación residen en tarjetas de memoria CF (se eligió este tipo de tarjeta por su robustez).

El Instituto RTS usa un sistema operativo basado en Linux con la extensión Xenomai. Por medio de Xenomai, todos los componentes RACK son capaces de ejecutar en el espacio de usuario y al mismo tiempo enlazando librerías como la *libraw* 1394 (*Firewire*) [15] o la *opencv* [16]. Los *drivers* de bajo nivel para el espacio del kernel se pueden acceder por medio del uso de la API *Real-Time Driver Model* para conformar otros *drivers.* Estos se enlazan por ejemplo con un bus CAN o con puertos serie. TiMS usa Rtnet [3] para enviar mensajes de tiempo real a través de Ethernet.

RACK soporta el desarrollo de componentes por medio de una librería de clases en C++. El servicio TiMS, el cual trabaja en el espacio del kernel fue implementado en C. Se provee además un paquete Java y un *framework* para desarrollar interfaces de usuario (incluyendo módulos *joystick*), o descargar componentes o realizar prototipos de componentes no críticos.

Fig. 8. Dos cajas de procesamiento escalable (SPB) con distintas capacidades periféricas, desarrolladas en el RTS.

4.1 . Servicio de mensajería TiMS

El servicio de mensajería TiMS (por sus siglas en inglés *Tiny Messaging System*) fue desarrollado en el RTS como parte de RACK. TiMS consiste de un módulo de kernel y una librería de espacio de usuario. Las funciones centrales de TiMS y de enrutado sobre Rtnet así como sobre TCP/IP son implementadas dentro del módulo kernel. Para usar el servicio de mensajes por medio de componentes en el espacio de usuario se provee una librería que define *mailboxes* y las funciones correspondientes. Un componente puede elegir entre tres modos para recibir mensajes:

- Bloqueando sin tiempo muerto o *timeout* (función *peek*)
- Bloqueando solo una cantidad de tiempo (función *peekTimed*)
- Sin bloquear (*peekIf*), lo que significa retornar inmediatamante en todos los casos.

La librería TiMS soporta dos opciones para obtener mensajes: los mensajes cortos son copiados durante una llamada a *receive*() en un *buffer* de datos dentro del receptor. El área de mensaje dentro del *mailbox* es liberada inmediatamente. Si se recibe un mensaje largo, el proceso de copiado puede ser muy largo y los recursos pueden ser bloqueados innecesariamente. En este caso, el área de memoria del mensaje puede ser bloqueada por la función *peek*() y así la tarea receptora puede trabajar directamente en su área de memoria. Cuando se termina todo el trabajo, el área es desbloqueada con la función *peek_end*().

TiMS obtiene todos los mensajes de RACK y los envía al *mailbox* del componente destino usando ya sea la pila de TCP/IP (tiempo no real) o Rtnet (tiempo real). Si el *mailbox* está localizado en el mismo *host*, TiMS copia el mensaje directamente al *buffer* de datos de un *mailbox* libre y despierta a la tarea que estaba aguardando en el componente correspondiente. La Fig. 9 muestra el flujo de datos.

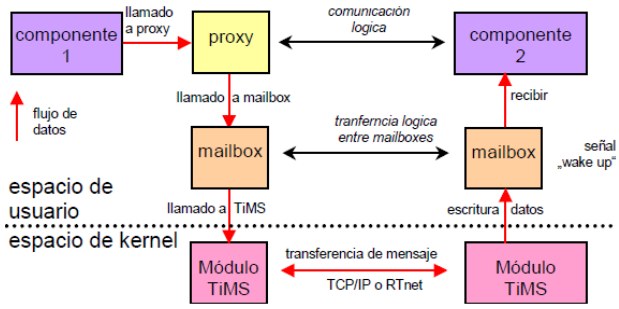

Fig.9. Implementación de la comunicación entre componentes.

Para enrutar mensajes de forma determinística, debe existir una conexión por cable de Ethernet. Los mensajes a la *interface* de usuario son transmitidos por la pila normal TCP/IP de Linux. Por lo tanto, TiMS posee un enrutador interno para reenviar todos los mensajes que no están dirigidos al sistema local hacia un enrutador en un nivel superior (2.2-3)

4.2 . Componentes

Los componentes RACK son aplicaciones independientes que pueden ser ejecutadas individualmente. Se implementan usando los lenguajes de programación C++ y Java. Todas las funciones de *interface* descritas en 2.4 se implementan como funciones virtuales en la clase *Module.*

Dentro de la implementación de la función *main()* solo se necesita una sobrecarga de las funciones virtuales para implementar el módulo. A continuación, se muestra el código fuente de un modulo simple:

```
class SimpleModule : Module {
        moduleInit() {
                Module::Init();
                allocate_ressources();
        }
        moduleCleanup() {
                free_ressources();
                Module::cleanup();
        }
        moduleOn()
                init_something();
                Module::moduleOn();
        }
        main(){
                simple =new SimpleModule();
                simple->moduleIni();
                return 0;
        }
}
```

La clase **DataModule** se deriva de la clase **Module** y extiende la clase con las funciones explicadas en la sección 2.5. Dentro de *init()* se reserva memoria para el *buffer* de datos y en la función *cleanup()* se libera nuevamente.

Por ejemplo, un modulo Scan2D se deriva de la clase *DataModule*. A continuación se muestra el código fuente de este módulo:

```
Class Scan2dModule : DataModule{
        LadarProxy ladar;
        RackMailbox ladarMbx;

        ModuleInit(){
                close_sensor_driver();
                delete_ladar();
                ladarMbx.destroy();
                DataModule::Cleanup();
        }

        //maneja los comandos proxy definidos
        moduleCommand(){
                if( recv_command == my_command) {
                        handle_command();
                }
                else {
                //atiende los comandos proxy
DataModule::moduleCommand(recv_command);
                }
        }

        moduleOn() {
                ladar->On();
                ladar->getContData(ladarMbx);
                DataModule::moduleOn();
        }

        moduleOff() {
                DataModule::moduleOff();
                ladar->stopContData(ladarMbx);
                ladar->Off();
        }

        moduleLoop() {
        buffer = get_local_buffer();
        ladarData = ladarMbx.recv();
        driverData = get_data_from_sensor();
        buffer = calculate_data(ladarData,
                        driverData);
        put_local_buffer();
        }
};
```

4.3 . Interface Gráfica de Usuario

La GUI está implementada en Java y es capaz de controlar todos los componentes de diferentes sistemas robóticos. Todos los componentes cargados son detectados y mostrados en la *interface* (Fig. 10). Estos pueden ser habilitados y deshabilitados. La información de los componentes es actualizada periódicamente con una prioridad mas baja que la de los trabajos críticos y es mostrada en la GUI en ventanas separadas. Cada clase RACK provee una extensión GUI para facilitar el despliegue de su propia información. Dentro de la *interface*, el servicio de registro de datos muestra toda la información en la ventana de mensajes. El concepto modular de RACK se hace explícito a través de la GUI.

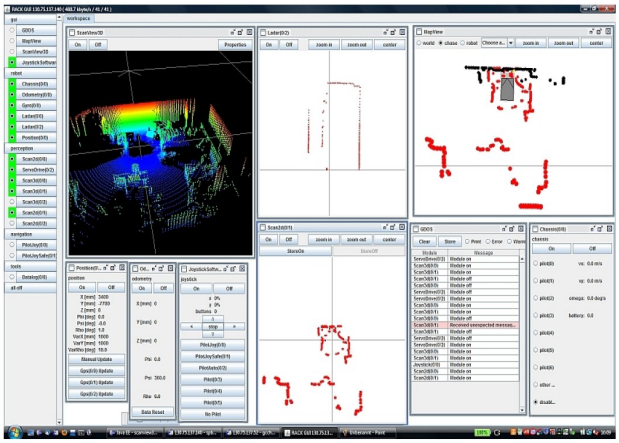

Fig. 10. Interfaz gráfica de usuario mostrando los módulos Scan3D, Scan2D y GDOS, entre otros.

4.4 . Servicio de Información

Para colectar y desplegar toda la información de los componentes distribuidos se usa el sistema genérico distribuido de salida (GDOS). Este envía todos los mensajes en el contexto de tiempo no real con la más baja prioridad a la *interface* central de usuario usada por el operador del sistema. El servicio muestra secuencialmente los mensajes de información y depurado de todos los componentes distribuidos en la GUI y convierte todas las direcciones de los *mailboxes* en nombres legibles (0x00012000 -> Ladar[20]).

La información desplegada es coloreada de acuerdo al nivel de información. De esta manera se pueden notar mensajes importantes al mismo tiempo.

El nivel puede ser dinámicamente fijado por la GUI para mantener libre el ancho de banda transmisión.

Existen cinco niveles de despliegue de información de depurado implementados: *print*, *warning*, *error*, *debug* y *debug_detail*. Todos los mensajes de información pueden ser grabados en un medio de almacenamiento para su posterior análisis.

5. SISTEMA ROBÓTICO DE EJEMPLO

En esta sección se describe uno de los sistemas robóticos autónomos del RTS, Hanna, el cual usa RACK. El vehículo robótico para ambientes exteriores tiene una longitud de 285 cm y un peso de 800 kg. Hanna puede navegar autónomamente a una velocidad máxima de 40km/h. El vehículo está equipado con una serie de componentes incluyendo módulos para control del chasis, sensores de rango láser 2D, GPS, acelerómetro, un sensor láser 3D de alta resolución, pilotos para navegación, planificador de caminos y localización. El robot se muestra en la Fig. 11.

Fig. 11. Robot Hanna.

Los datos del sensor láser 2D son adquiridos por el componente Scan2D con el fin de generar mapas laser en 2 dimensiones (2D *laser scan*s), mientras que el componente Scan3D genera nubes de puntos en tres dimensiones (información tridimensional sobre el entorno). Cada mapa es un conjunto de puntos correspondientes a las distancias medidas por el sensor. Estas son requeridas por un módulo que usa el método de localización *Monte Carlo* [5] así como por el módulo *PilotPath* para calcular la posición actual y detectar obstáculos en el rango visible.

El módulo piloto también obtiene datos del componente *Path*, el cual crea trayectorias curvilíneas (*splines*) por medio de puntos de ruta dados por el usuario y envía comandos al módulo *ChasisHanna* del robot móvil. El componente para localización por el método Monte Carlo requiere un mapa de todos los obstáculos estáticos en el área de navegación para asegurar un posicionamiento exacto.

La prioridad dada a los componentes en ejecución depende de la respuesta de tiempo necesaria. La detección de obstáculos en el rango del robot tiene que ocurrir lo mas rápido posible. De esta manera a un componente que tiene que reaccionar muy rápidamente se le asigna una alta prioridad. Los componentes que planifican nuevos caminos hacia los objetivos dados tienen una prioridad mas baja. En la Fig. 12 se muestra el flujo de datos, así como el período usado y la prioridad dada a los componentes RACK ejecutándose en el robot Hanna.

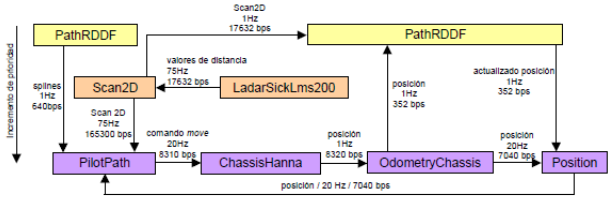

Fig.12. Flujo de datos y priorización para el robot Hanna.

6. CONCLUSIONES Y PERSPECTIVAS

En este artículo se presentó el *framework* RACK el cual facilita y optimiza el desarrollo de sistemas robóticos distribuidos. RACK incluye todos los componentes abstractos de interfaz con los sensores y actuadores robóticos típicos, así como con componentes genéricos de procesamiento de datos. Un amplio rango de tareas es cubierto por medio del soporte del desarrollo de componentes de tiempo real y no real. El sistema de mensajería TiMS maneja la comunicación entre los componentes; TiMS es capaz de satisfacer tanto restricciones

estrictas de tiempo como requerimientos de desempeño.

Cabe recalcar que el *framework* se puede utilizar en otras ramas de la ingeniería, como por ejemplo el control de procesos, donde se tendría en lugar de componentes asociados a sensores láser, GPS u odometría, componentes asociados a la lectura de sensores de temperatura o presión y, en vez de *drivers* para manejar el chasis, módulos para controlar válvulas o ventiladores. La filosofía del sistema se mantendría intacta, lo que cambia son las rutinas de los *drivers* (manejo) y de los controladores.

Dentro del alcance de proyectos de investigación se desarrollaron varios componentes robóticos o se migraron componentes del *framework* anterior. Un número significativo de esos módulos esta disponible actualmente como software de código abierto. Algunas mejoras futuras al sistema RACK incluyen:

Soporte mejorado en tiempo no real: a fin de facilitar la integración de algoritmos complejos no determinísticos, los componentes podrán también ser ejecutados en tiempo no real. Esto incluirá un soporte por parte de TiMS para sistemas Linux estándar.

Control estricto de acceso: un proyecto actual del RTS presta atención a los asuntos de seguridad del servicio de mensajería TiMS [4]. El objetivo es restringir el número de componentes críticos RACK lo cuales pueden incluir código malicioso. Esta restricción se hará controlando los tipos admisibles de mensajes, sus prioridades y el ancho de banda usado, de manera similar a la solución propuesta de protección para el Rtnet [5].

Compresión de Mensajes: ya que algunos de los datos de la información sensorial contienen una redundancia significativa, se debe incluir una compresión de mensajes simple pero rápida en el servicio TiMS. Esto ayudará a reducir adicionalmente los requerimientos de ancho de banda, específicamente en el caso de enlaces inalámbricos.

Herramienta de configuración: Para facilitar la configuración de sistemas robóticos, la cual incluye la distribución y parametrización de los componentes, así como la asignación de recursos de comunicación se planea una herramienta gráfica de soporte basada en el entorno de desarrollo Eclipse [23]

RACK es un software libre, disponible bajo la licencia publica general reducida (LGPL). Mas aún, muchos de estos componentes están basados en librerías externas de código abierto. El proyecto pretende fomentar el desarrollo de componentes adicionales de código abierto y la reutilización de soluciones libres disponibles. Al mismo tiempo, este permite construir proyectos de software propietario basados en él.

Para descargas y mayor información, se puede visitar la página:

http://developer.berlios.de/projects/rack

7 . REFERENCIAS BIBLIOGRÁFICAS

[1] S. Thrun, M. Beetz, et al., *Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot*, International Journal of Robotics Research, Vol 19(11):972999, 2000.

[2] O. Wulf, J. Kiszka, B. Wagner, *A Compact Software Framework for Distributed Real-Time Computing*, in 5th Real-Time Linux Workshop, Valencia, Spain, 2003.

[3] J. Kiszka, B. Wagner, Y. Zhang, J. Broenink, RTnet – A Flexible Hard Real-Time Networking Framework, 10th IEEE International Conference on Emerging Technologies and Factory Automation, Catania, Italy, disponible en: http://www.rts.uni-hannover.de/rtnet, 2005.

[4] Kiszka, J. and B. Wagner, *Domain and Type Enforcement for Real-Time Operating Systems*, 9th IEEE International Conference on Emerging Technologies and Factory Automation, Lisbon, Portugal, 16-19 September 2003.

[5] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, Monte Carlo Localization for Mobile Robots, IEEE International Conference on Robotics and Automation (ICRA99), 1999.

[6] D.C. Schmidt, F. Kuhns, An Overview of the Real-time CORBA Specification, IEEE computer special issue on Object-Oriented Real-time Distributed Computing, 2000.

[7] P. Hohmann, U. Gerecke, B. Wagner, A Scalable Processing Box for Systems Engineering Teaching with Robotics, International Conference on Systems Engineering, Coventry, UK, 2003.

[8] OROCOS, Open Robot Control Software, disponible en: www.orocos.org, consultado en 2006.

[9] Object Management Group, The Common Object Request Broker: Architecture and specification (CORBA 2.4.1), disponible en: www.omg.org, consultado en 2006.

[10] D. Austin, Dave's Robotic Operating System, disponible en: www.dros.org, consultado en 2010.

[11] The Player/Stage Project, disponible en: http://playerstage.sourceforge.net, consultado 2005.

[12] Mobile and Autonomous Robotics Integration Environment, disponible en: http://marie.sourceforge.net/index.html, 2006.

[13] P. Gerum, Xenomai: Real-time framework for Linux, disponible en: http://www.xenomai.org, consultado en 2006.

[14] P. Mantegazza, E. Bianchi, et al., RTAI: Real-Time Application Interface, Linux Journal, disponible en: www.rtai.org, consultado en Septiembre de 2000.

[15] IEEE 1394 userspace library, disponible en: http://sourceforge.net/projects/libraw1394, 2010.

[16] Open Source Computer Vision Library, disponible en: www.intel.com/technology/computing/opencv, consultado en Septiembre de 2010.

[17] D. Lecking, O. Wulf, B. Wagner: *Localization in a wide range of industrial environments using relative 3D ceiling features*, 13th IEEE International Conference on Emerging Technologies and Factory Automation, Hamburg, 2008.

[18] SICK AG, disponible en: http://www.sick.com/home/en.html, 2006.

[19] IBEO Automobile Sensor GmbH, 2006, http://www.ibeoas.de/html/prod/prod.html.

[20] Robosoft - Advanced Robotic Solutions, disponible en: http://www.robosoft.fr/outdoor-robot.html, consultado en 2003.

[21] A. Birrell, B.J. Nelson, *Implementing remote procedure calls*, ACM Transactions on Computer Systems, Vol 2, pp. 39-59, 1984.

[22] P.T. Eugster, P.A. Felber, R. Guerraoui, A-M. Kermarrec, *The many faces of publish subscribe,* ACM Comput. Surv., 35(2):114-131, 2003.

[23] Eclipse Development Tools, disponible en: http://www.eclipse.org/, 2010.