Un Framework de Desarrollo para Robótica

L. Manso, P. Bustos, P. Bachiller, P. Núñez, R. Cintas and L. Calderita

***Resumen*—Este artículo presenta RoboComp, un framework libre de desarrollo de software para robots. Se presenta también una comparación con los proyectos similares más relevantes, especificando sus bondades y debilidades.**

La experiencia del usuario final es uno de los aspectos más importantes a tener en cuenta cuando se desarrolla software para robots. En la práctica, las herramientas tienen una importancia radical a la hora de desarrollar componentes, por lo que el artículo describe en profundidad las utilidades que hacen de RoboComp más que un middleware para robótica. A lo largo del texto se muestran distintos ejemplos de uso con el fin de demostrar la experiencia final del usuario.

***Palabras clave*—robocomp, middleware, robótica.**

I. Introducción

CUANDO se desarrolla software para robots se tienen que abordar problemas muy específicos. Además, ha de ser a la vez eficiente, fácil de utilizar y de extender. Para poder conseguir buenos resultados se deben seguir técnicas concretas de ingeniería del software, que aborden las siguientes cuestiones: a) complejidad conceptual del software, b) reusabilidad y escalabilidad del código, c) distribución de la computación, d) soporte multiplataforma y multilenguaje, e) independencia del hardware.

La complejidad del software, desde el punto de vista del desarrollador, es muy importante ya que la escalabilidad disminuye al aumentar la complejidad. La comunidad científica es consciente de este hecho y en los últimos años se ha orientado claramente hacia la programación orientada a componentes. Los *componentes* son programas de ejecución independiente que proporcionan una interfaz que otros componentes pueden usar. Son, un concepto más abstracto que el de *clase* (de hecho los componentes suelen estar constituidos por varias clases). Esto hace que usar componentes facilite la comprensión de sistemas complejos, porque para entender el sistema globalmente no hace falta conocer las peculiaridades de los componentes, sólo su funcionalidad. Sin embargo, a pesar de que existe un aparentemente consenso en el uso de programación orientada a componentes, se han propuesto diversas tecnologías de desarrollo.

En este artículo se presenta RoboComp, un framework orientado a robótica que aporta unas importantes características técnicas sin perder de vista la sencillez de uso. Uno de los factores clave del diseño de RoboComp es que, al igual que Orca2[8], está basado en Ice[1]. Esta decisión no sólo evita invertir tiempo en desarrollar un middleware ad-hoc, sino que también libera de las correspondientes comprobaciones y mantenimiento. Ice es un middleware liviano y de calidad industrial que aporta gran fiabilidad (ha sido usado en varios proyectos críticos [2]). Además facilita la programación de componentes robustos gracias a que el marshalling de datos es automático (hacer esto a mano es propenso a errores) y a que ofrece mecanismos de lanzamiento de excepciones entre componentes (emitidas automáticamente por fallos en la comunicación o por los propios componentes remotos).

La mayor aportación de RoboComp es el conjunto de herramientas que ofrece, así como su sistema de instalación. Estas herramientas, detalladas en la sección III, hacen que la programación sea más sencilla y ágil.

RoboComp proporciona una amplia variedad de componentes. Existen, por ejemplo, componentes dedicados a interfaz hardware (p.e. cameraComp, differentialRobotComp o laserComp), a la implementación de comportamientos (p.e. gotopointComp o wanderComp) o al procesamiento de datos (p.e. visionComp y roimantComp para detección de características visuales, y cubafeaturesComp para la detección de características láser). RoboComp también dispone de una amplia documentación online que cubre todos los aspectos que los usuarios puedan necesitar. Como se verá a lo largo del artículo, la creación de nuevos componentes y su integración con otros ya existentes es altamente intuitiva.

II. Características principales

Como se ha señalado en la introducción, en los últimos años se han propuesto varios frameworks de robótica. Los que han sido considerados más relevantes, y por tanto se han incluido en la comparación (ver sección IV) son: Carmen[3], Marie[5], Miro[6], MOOS[7], Orca2[8], OROCOS[9], Player[10], ROS[11] o YARP[12]. Esta sección, que tiene el fin de ofrecer una descripción pormenorizada de RoboComp, describe sus principales características.

A. Características de comunicación

Gracias a que RoboComp trabaja sobre Ice[1], el equipo de desarrollo se ha ahorrado una gran cantidad de trabajo no sólo asociada a la creación de un middleware ad-hoc, sino también a su futuro mantenimiento. Elegir Ice es una decisión acertada puesto que reune los requisitos que experimentalmente se atribuyen a un framework para robótica:

- Diferentes métodos de comunicación (e.g. RMI, pub/sub, AMI and AMD).
- Basado en IDL, con interfaces fuertemente tipados.
- Buen rendimiento con poca sobrecarga[1].
- Comunicaciones eficientes.
- Soporte multiplataforma y multilenguaje.

L. Manso, P. Bustos, P. Bachiller, P. Núñez, R. Cintas and L. Calderita son miembros del grupo RoboLab. Universidad de Extremadura.
E-mail: lmanso@unex.es

Este trabajo ha sido realizado gracias a la financiación poropoccionada por la Consejeria de Economía, Comercio e Innovación de la Junta de Extremadura (PRI09A037), y por el Gobierno de España y los fondos FEDER (TSI-020301-2009-27).

Dado que la reutilización de un middleware conlleva el ya mencionado ahorro de tiempo (que puede ser destinado a otras tareas) y que Ice cubre todos los requisitos sobradamente, se decidió basar RoboComp en Ice. Esta elección, además, mejora la compatibilidad con otros frameworks basados en Ice como Orca2[8] o JDE[4].

Todas las características de comunicación de RoboComp son, por lo tanto, heredadas de Ice. Soporta una amplia variedad de plataformas y lenguajes de programación. También tiene en cuenta cuestiones relacionadas con la seguridad: opcionalmente, los componentes pueden ser autenticados y las transmisiones pueden ser cifradas. Además, Ice ha sido usado en varios proyectos críticos[2], por lo que puede ser considerado como un middleware muy maduro. Soporta diferentes métodos de comunicación: suscripción, invocación de métodos remotos, AMI y AMD.

B. Soporte de lenguajes

Hoy en día, los robots se desarrollan por equipos de personas muy heterogéneos que pueden no utilizar el mismo lenguaje de programación. De hecho, la elección de un lenguaje de programación puede variar en función a la tarea a realizar. El soporte de diferentes lenguajes es, por lo tanto, una característica muy deseable. Los lenguajes soportados por Ice, y por extensión RoboComp también, son: C++, C#, Java, Python, Objective-C, Ruby y PHP.

Debido a que la creación de software para robótica es una tarea muy compleja en sí misma, el desarrollo no debería verse dificultado por la programación de mecanismos de comunicación entre componentes. Ice proporciona una sencilla interfaz orientada a objetos. Con ella, por ejemplo, la invocación de métodos remotos, se mantiene la misma sintaxis que la de métodos de objetos locales. Esto no sólo simplifica la estructura del código, sino que también permite al desarrollador centrarse en el problema concreto a resolver.

C. Herramientas y clases

Las herramientas de RoboComp y su conjunto de clases son dos de los aspectos fundamentales en los que RoboComp extiende a Ice. Éstas mejoran la usabilidad, y hacen más sencillo el desarrollo de sistemas complejos. Ya que las herramientas son la contribución más importante, se describen ampliamente en la sección III. El conjunto de clases comprende diferentes aspectos relacionados con la robótica y la vision artificial como el cálculo matricial, el acceso al hardware, filtros de kalman, widgets gráficos, lógica difusa o la propiocepción del robot. Entre las diferentes clases disponibles, la funcionalidad de la clase de propiocepción, a la que llamamos *InnerModel*, merece una mención especial por el importante papel que tiene en los robots. *InnerModel* trabaja en uno de los aspectos más propensos a errores y difíciles de depurar: la representación física del robot y las transformaciones geométricas entre diferentes sistemas de referencia.

D. Características HAL

La programación orientada a componentes es una forma eficaz de mantener la complejidad del software bajo control. Sin embargo, la mayoría de los frameworks para robótica han ignorado una de las principales aportaciones de Player[10]: una capa HAL es extremadamente importante para la reusabilidad y portabilidad del código. La mayoría de los sensores y actuadores del mismo tipo, a no ser que tengan características muy específicas, pueden tener una interfaz común. Esto es preferible siempre que sea posible por los siguientes motivos: a) diferentes usuarios, investigadores o empresas pueden compartir sus componentes independientemente del hardware usado,b) facilita la localización de errores: si un usuario cambia el hardware de un robot y algún componente de alto nivel deja de funcionar, el componente encargado del acceso al hardware es el causante del fallo, c) alarga la vida útil de los componentes, quedando acotada por el tiempo de vida de las interfaces utilizadas.

Ante la evidencia de la existencia de hardware que no se puede ajustar a un interfaz estándar, RoboComp es flexible en relación a esta cuestión: la utilización de los interfaces propuestos no es obligatoria sino *aconsejable*. Proponer una capa HAL no significa que exista, por ejemplo, un único componente *cameraComp* que trabaje con todas las posibles cámaras existentes. Significa que existe un fichero IDL que especifica una interfaz que los distintos componentes que proveen imágenes pueden ofrecer y que los componentes que las necesitan pueden usar. Actualmente, RoboComp tiene definidas las siguientes interfaces HAL: Camera, DifferentialRobot, GPS, IMU, Joystick, Laser, Micro y JointMotor.

E. Organización del Framework

RoboComp está organizado jerárquicamente con el fin de hacerlo más sencillo de entender y extender. La organización básica se muestra en la figura 1.

Los directorios *Classes* y *Tools* contienen varios subdirectorios con diferentes clases y herramientas, respecetivamente. *CMake* contiene los ficheros de CMake, con los que se consigue un sistema de compilación robusto y portable. *Interfaces* contiene los ficheros que definen las diferentes interfaces. *Components* contiene el directorio *HAL* (donde los componentes HAL están ubicados) y un subdirectorio de componentes desarrollados por nuestro grupo de investigación (RoboLab) en el que están almacenados los componentes que no encajan dentro de HAL. Esta es la estructura del código

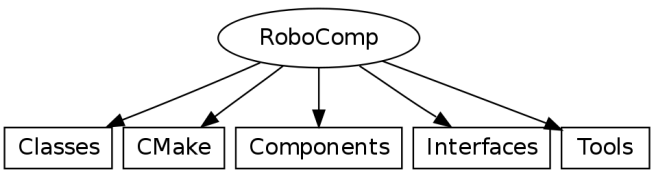

Fig. 1. Jerarquía de directorios RoboComp.

fuente de RoboComp. Sin embargo, los usuarios no tienen que adherirse necesariamente a esta estructura. Los nuevos componentes pueden ser distribuidos por separado sin ningún tipo de problema. Esto es posible gracias a dos variables de entorno: *$ROBOCOMP* (que contiene la ruta del código fuente de RoboComp) y *$SLICE_PATH* (una lista que contiene las

posibles rutas en las que los ficheros de interfaz pueden ser encontrados).

III. HERRAMIENTAS

Los aspectos a tener en cuenta en un framework de robótica no son sólo técnicos. También son muy importantes factores como la facilidad de uso o la adaptación a ciclos de desarrollo ágiles. Con el fin de ofrecer estas propiedades, RoboComp cuenta con diferentes herramientas que complementan la capa middleware.

A. componentGenerator

Los componentes de RoboComp mantienen una estructura similar. Generalmente se componen de tres elementos principales: la clase interfaz, la clase *worker* y los proxies de comunicación con otros componentes. La clase *worker* es la que implementa la funcionalidad propia del componente. La clase interfaz hereda de una clase virtual autogenerada a partir de la interfaz definida en el fichero Slice (el IDL de Ice). Los métodos que tiene que implementar son los métodos que define la interfaz Ice. Durante la ejecución del componente, ésta se encarga de responder a las llamadas. Los proxies son también clases autogeneradas que dan acceso a objetos remotos, permitiendo la comunicación con otros componentes.

Dado que la mayoría de los componentes mantienen la misma estructura, el proceso de creación de nuevos componentes es automatizable. Por ello, se creó la utilidad *componentGenerator*. Su salida es un árbol de directorios con el código de conexión a otros componentes ya incluido, de manera que los usuarios se pueden concentrar en la funcionalidad real del componente (generalmente implementar la clase worker y otras que se puedan necesitar).

Para mostrar el pequeño número de líneas que el usuario ha de introducir, se muestra como ejemplo la creación de dos componentes que interactúan para resolver la suma de dos números. Los listados 1, 2, 3 y 4 corresponden a los ficheros a modificar. El componente cliente lee dos números por teclado y solicita al componente servidor el resultado de la suma (generalmente no tiene por qué haber diferenciación entre componente cliente y servidor ya que pueden desempeñar ambas roles). El componente servidor espera las peticiones de los clientes y responde mediante la clase interfaz. Ambos componentes fueron creados con la herramienta *componentGenerator*.

El listado 1 muestra el único fichero a modificar en el componente cliente. El resto de los ficheros del cliente no se muestran debido a que son autogenerados y no necesitan sufrir ningún cambio. En el código del listado 1 se muestran en negro y marrón (línea 12) las líneas incluidas por el programador. El resto de líneas son autogeneradas. En concreto, la línea en marrón muestra la invocación del método remoto en el componente servidor. El uso de una llamada RPC mediante un proxy permite al programador usar con un objeto remoto la misma sintaxis que para un objeto local.

El los listados que van del 2 al 4 se pueden ver los ficheros fuente del componente servidor que se tienen que modificar para añadir la funcionalidad específica del componente. Al igual que en el listado 1, sólo las líneas resaltadas son escritas por el programador.

Listado 1. *worker.cpp* del cliente

```
1  #include "worker.h"
2
3  Worker::Worker(AddTwoIntsServerPrx
       addtwointsserverprx, QObject *parent) : QObject(
       parent) {
4    addtwointsserver = addtwointsserverprx;
5    connect(&timer, SIGNAL(timeout()), this, SLOT(
       compute()));
6    timer.start(BASIC_PERIOD);
7  }
8
9  void Worker::compute() {
10   int a, b, res;
11   cin>>a;  cin>>b;
12   res = addtwointsserver->addTwoInts(a,b);
13   cout<<"a+b="<<res<<endl;
14 }
```

Listado 2. *AddTwoIntsServer.ice*

```
1  #ifndef ADDTWOINTSSERVER_ICE
2  #define ADDTWOINTSSERVER_ICE
3
4  module RobolabModAddTwoIntsServer {
5    interface AddTwoIntsServer {
6      int addTwoInts(int a, int b);
7    };
8  };
9
10 #endif
```

Listado 3. *AddTwoIntsServerI.cpp*

```
1  #include "AddTwoIntsServerI.h"
2
3  AddTwoIntsServerI::AddTwoIntsServerI( Worker *_worker
       , QObject *parent) : QObject(parent) {
4    worker = _worker;
5    mutex = worker->mutex;
6  }
7
8  int AddTwoIntsServerI::addTwoInts(int a, int b, const
       ::Ice::Current&) {
9    return(worker->addTwoInts(a,b));
10 }
```

Listado 4. *worker.cpp* del servidor

```
1  #include "worker.h"
2
3  Worker::Worker(QObject *parent) : QObject(parent) {
4    mutex = new QMutex;
5    connect(&timer, SIGNAL(timeout()), this, SLOT(
       compute()));
6    timer.start(BASIC_PERIOD);
7  }
8
9  void Worker::compute( ) {
10 }
11
12 int Worker::addTwoInts(int a, int b)  {
13   return(a + b);
14 }
```

El listado 2 es la definición Slice, donde se especifica la interfaz que cada componente ofrece. El listado 3 muestra la implementación de dicha interfaz. Es esta implementación (derivada de la clase definida por la interfaz Ice) la encargada de gestionar las peticiones de los clientes. Esto se lleva a cabo generalmente mediante la interacción con la clase *worker*. Además, en el caso del cliente, la clase *worker* (listado

4) incluye la implementación del núcleo del componente. Como se puede observar, en el componente servidor sólo se han de incluir las líneas relacionadas con la definición e implementación de servicios, sin tener en cuenta aspectos de bajo nivel.

B. *managerComp*

Los componentes son programas que, a pesar de ejecutarse independientemente, interactúan entre sí. En principio se pueden ejecutar manualmente desde consola, pero cuando interviene un cierto número de ellos la tarea puede resultar tediosa. Con el fin de evitar esta situación se ha desarrollado un gestor de componentes *managerComp* (figura 2). Esta

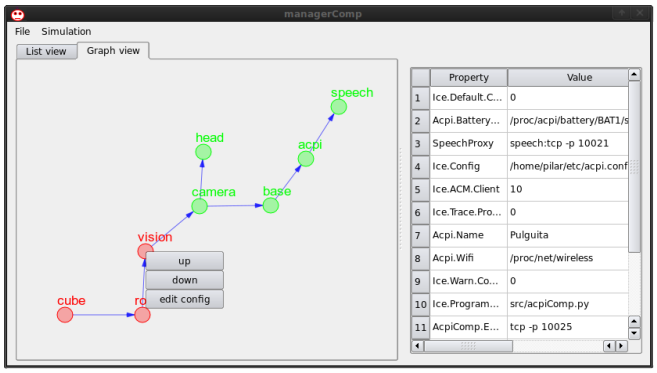

Fig. 2. Interfaz de *managerComp*.

herramienta utiliza un sistema de configuración XML en el que cada componente a gestionar se indica dentro de un nodo XML. Para cada nodo asociado a un componente se tienen que especificar, además de un alias, su endpoint (una cadena que especifica cómo conectarse a él), los comandos para arrancarlo y pararlo, y los componentes de los que depende. Con esta información, *managerComp* puede comprobar si las dependencias de un componente se encuentran en ejecución (si no es así, intenta arrancarlas). De esta forma, cuando un usuario solicita la ejecución de un componente mediante esta herramienta, sus dependencias se satisfacen automáticamente. Además, mediante SSH, *managerComp* pueden también ser usado para manejar componentes remotos prácticamente de la misma forma que los locales.

C. *monitorComp*

Desarrollar nuevos componentes conlleva, en algún momento, la evaluación de su funcionamiento. Sin una herramienta específica, se tendría que crear un nuevo componente sólo para realizar esta tarea. Para evitar esto, RoboComp cuenta con *monitorComp*, una herramienta para la monitorización de componentes. Permite al usuario especificar pequeños scripts Python que realizan las comprobaciones pertinentes con un número muy reducido de líneas de código. De hecho, con RoboComp se distribuyen diversos test que pueden ser utilizados como plantillas.

Para utilizar esta herramienta, el primer paso es especificar la conexión al componente a monitorizar: se ha de especificar el endpoint y el fichero Slice que define la interfaz del componente. Una vez que se han introducido los datos de conexión, el siguiente paso es la introducción del código de monitorización (figura 3). Para facilitar este paso, la herramienta muestra tanto los símbolos definidos en el fichero Slice, como los métodos remotos disponibles. El lenguaje elegido tanto para la aplicación como para los scripts es Python debido a su gran capacidad de introspección y a su expresividad (con la que se reduce el número de líneas necesarias para los tests).

Una vez que el código está escrito, *monitorComp* puede lanzar pruebas sin necesidad de ninguna entrada del usuario. Toda la información necesaria puede ser leída desde fichero. La figura 3 muestra un ejemplo de la especificación del código de prueba. Con pocas líneas basta para recuperar y mostrar las imágenes de un par de cámaras estéreo.

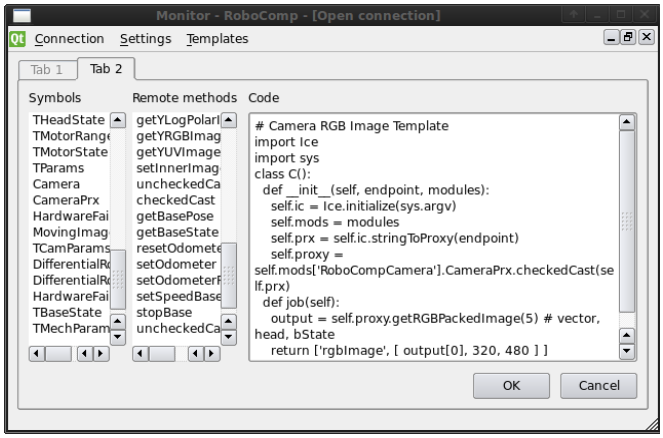

Fig. 3. Monitoring code insertion.

D. *replayComp*

La herramienta *replayComp* (imágen 4) se encarga de almacenar la salida del hardware para posteriormente reproducirla. Durante la reproducción, se puede especificar la velocidad de avance de la grabación o avanzar paso a paso. Esta herramienta es extremadamente útil durante el proceso de depuración cuando se pretende reproducir errores: si la entrada es la misma, el comportamiento de los programas debería ser también el mismo. También es muy útil para evitar que la no disponibilidad física de un robot detenga el desarrollo del software. Esta herramienta funciona en dos modos: captura

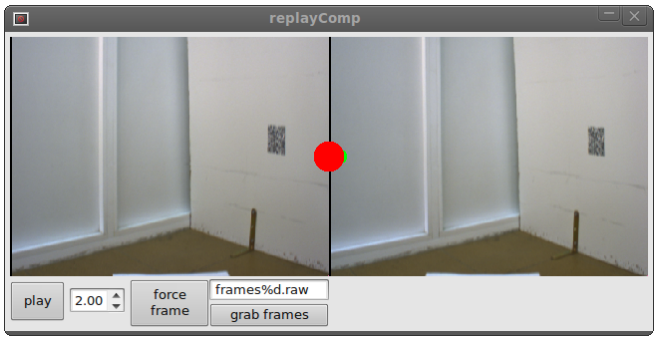

Fig. 4. Interfaz de *replayComp*.

y reproducción. En el modo captura, *replayComp* se conecta a los componentes HAL y graba sus salidas. En modo de reproducción este fichero es utilizado como entrada para otros componentes. Esto se realiza de forma totalmente transparente, es decir, los componentes que reciben los datos de replayComp no son capaces de discernir entre replayComp y el hardware real. La herramienta implementa las interfaces de todos los componentes HAL, proporcionando la capa de hardware cuando el robot no está fisicamente disponible. Gracias a que las interfaces son las mismas, los componentes se pueden conectar a replayComp simplemente cambiando su configuración, sin ninguna modificación del código.

Como se ha comentado anteriormente, replayComp implementa todas las interfaces HAL: DifferentialRobot, Laser, Camera, CamMotion, Micro y GPS.

E. Soporte de Gazebo

Un componente de reproducción es muy útil para la depuración del software que no requiera de un robot activo. Sin embargo, durante la depuración de comportamientos activos, un componente de reproducción no es suficiente. Con un robot real, la reproducción del comportamiento del software es una tarea útil pero muy compleja. Adicionalmente, en las primeras fases de desarrollo de un componente, puede ser de gran ayuda probar su funcionamiento en condiciones ideales, antes de pasar al complejo mundo real. Para aunar estas características, un simulador de robótica parece una herramienta muy adecuada. Así, RoboComp incluye soporte para Gazebo (figura 5), un simulador de robots en 3D libre. Como se ha mencionado antes, es muy útil en tareas de depuración, pero no sólo en ellas. También permite a los desarrolladores esquivar las limitaciones hardware o su ausencia durante la creación de nuevo software.

Para hacer la utilización del simulador transparente al software, el soporte para Gazebo ha sido incluido extendiendo los componentes HAL. Como resultado, los componentes de alto nivel no necesitan sufrir ningún cambio para funcionar sobre el simulador o sobre el robot real.

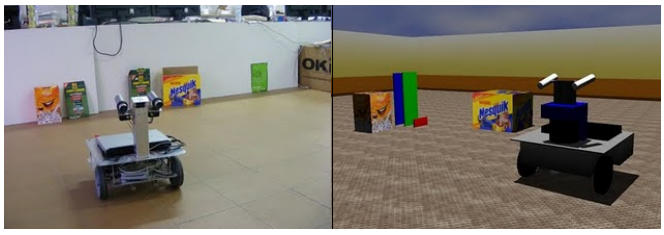

Fig. 5. Un robot RobEx y su simulación sobre Gazebo.

F. loggerComp

Esta herramienta es una alternativa opcional a la salida estándar, diseñada para recuperar y analizar los mensajes de texto emitidos por los componentes. Proporciona una interfaz gráfica para mostrar la información de interés (figura 6), permitiendo al usuario filtrarla por: tipo, componente, fecha de emisión, fichero y número de línea. Para utilizar loggerComp, los componentes sólo han de usar el proxy *loggerComp* e instanciar una clase que se encarga de usarlo: *qLog*.

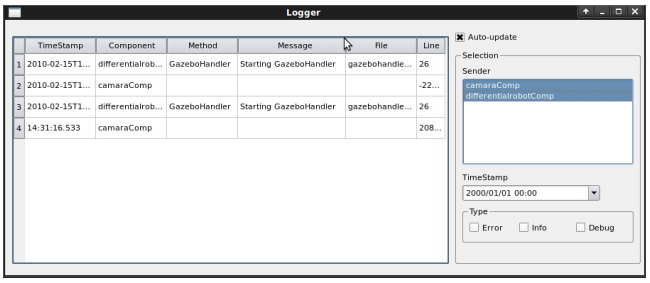

Fig. 6. Interfaz de *loggerComp*.

IV. Comparación de Frameworks

A lo largo del texto se han discutido diferentes características de los frameworks de robótica. Dependiendo de la tarea, será preferible utilizar distintas plataformas o lenguajes de programación. Para adecuarse a los diferentes eventos y patrones de comunicación es conveniente disponer de diferentes métodos de comunicación. Debido a la complejidad de los problemas a resolver, en la práctica, las herramientas son una de las características más importantes a tener en cuenta cuando se evalúa un framework de robótica. Con la intención de proporcionar una visión general comparativa de las características de RoboComp, esta sección presenta una comparación con los principales frameworks existentes.

La tabla I muestra la licencia, plataformas y lenguajes soportados por los diferentes frameworks considerados más relevantes. La tabla II muestra el middleware utilizado por los diferentes frameworks y algunas de sus características asociadas. Aquellos frameworks que proporcionan una interfaz fuertemente tipada basada en IDL son preferibles: son más sencillos de entender y, por tanto, en la práctica reducen la posibilidad de errores. En RoboComp y Orca2, se utilizan las herramientas de compilación de Ice para convertir las interfaces IDL en código específico del lenguaje de programación.

Tabla I
Aspectos generales

Framework	Licencia	Plataformas soportadas	Lenguajes de programación
Carmen	GPL	Linux, Windows	C, C++, Java, Python
Marie	LGPL	Linux	C, C++
Miro	GPL	Linux	C, C++
MOOS	GPL	Linux, Mac OS X	C++, Matlab
Orca2	LGPL/GPL	Linux, Windows, Mac OS X, Android, iPhone,	C++, C#, Python, PHP, Ruby, Java, Objective-C
OROCOS	LGPL/GPL	Linux, Windows	C++
RoboComp	GPL	Linux, Windows Mac OS X, Android, iPhone	C++, C#, Python PHP, Ruby, Java, Objective-C
ROS	BSD	Linux, Windows	C++, Python, Octave, Lisp
YARP	GPL	Linux, Windows, QNX	C++, Java, Ruby, Python, Lisp

Otros frameworks que utilizan el IDL de CORBA están equipados con herramientas similares. El resto de frameworks no describen su interfaz mediante IDL, aunque ROS sí proporciona un lenguaje de descripción de datos y servicios con sus herramientas de compilación correspondientes. Respecto a los métodos de comunicación, los frameworks basados en Ice proporcionan una buena variedad de mecanismos síncronos y asíncronos. Esta característica permite al programador seleccionar los métodos de comunicación entre componentes que mejor se ajusten a los tipos de interacciones que deben llevarse a cabo.

Tabla II
CARACTERÍSTICAS DE RED Y SU API

Framework	Middleware	Comm. Methods	IDL
Carmen	IPC	pub/sub, query/response	No IDL
Marie	ACE	data-flow (socket-based)	No IDL
Miro	ACE+TAO	sync/async, client/server	CORBA IDL
MOOS	propio	pub/sub	No IDL
Orca2	Ice	RPC, AMI, AMD, pub/sub	Ice IDL
OROCOS	ACE	commands, events, methods, properties, data-ports	CORBA IDL
RoboComp	Ice	RPC, AMI, AMD, pub/sub	Ice IDL
ROS	propio	pub/sub, query/response	No IDL
YARP	ACE	asynchronous data-flows	No IDL

Tabla III
COMPARATIVA DE LAS HERRAMIENTAS DISPONIBLES

Framework	Generador	Manager	Replay	Simulador	Log	Monitor
Carmen	no	no	gui	propio, 2d	gui	no
Marie	no	no	no	gazebo	no	no
Miro	no	no	no	no	gui	no
MOOS	no	texto	gui	uMVS	gui	no
Orca2	no	no	gui	gazebo	no	no
OROCOS	no	texto	no	no	no	no
RoboComp	text	gui	gui	gazebo	gui	gui
ROS	partial	texto	gui	gazebo	gui	gui
YARP	no	gui	no	icubSim	texto	no

La tabla III muestra las herramientas proporcionadas por los distintos frameworks. En ella, *texto* significa que la herramienta se ofrece en modo consola, *gui* corresponde a una interfaz gráfica, y *no* a la ausencia de dicha herramienta. La columna *'generador'* representa la disponibilidad de una herramienta de generación de código. A pesar de que tanto RoboComp como ROS proporcionan esta capacidad, sólo RoboComp produce código listo para usar: la herramienta de ROS *roscreate_pkg* crea el árbol de directorios de un paquete ROS así como los ficheros necesarios para su compilación, sin embargo, no genera código fuente listo para ser usado. La columna *'manager'* representa la existencia de una herramienta específica y sencilla para ejecutar y supervisar la ejecución de los componentes (Orca2 utiliza IceGrid, pero ni es una herramienta específica ni amigable). Las columnas *'replay', y 'log'* representan la disponibilidad de herramientas para recrear el comportamiento de componentes y para registrar mensajes, respectivamente. La columna *'simulador'* contiene los simuladores con los que los diferentes frameworks pueden trabajar. Permitir a los componentes trabajar con un simulador sin necesidad de introducir cambios en el código es una herramienta extremadamente útil, sobre todo en las fases más tempranas del desarrollo de un componente. RoboComp proporciona un uso del simulador Gazebo de forma totalmente transparente. La columna *'monitor'* representa la disponibilidad de una herramienta para monitorizar la información con la que está trabajando un componente, no sólo su estado (p.e. imágenes de una cámara, objetos detectados, o la posición de un robot). Tanto ROS como RoboComp proporcionan esta herramienta. Sin embargo, *monitorComp* es más avanzada que la herramienta con la que viene equipado ROS: *rxbag* tiene un sistema de plugins que permite al usuario mostrar nuevos tipos de datos, pero sólo es capaz de trabajar con datos off-line; *rxplot* trabaja con datos on-line pero sólo con una pequeña cantidad de tipos. *monitorComp* posee ambas características.

V. CONCLUSIONES Y TRABAJOS FUTUROS

En este artículo se ha descrito RoboComp, haciendo especial hincapié en sus objetivos, las decisiones de diseño tomadas, y las herramientas que ofrece.

RoboComp destaca en las comparaciones con otros frameworks en varios aspectos como la sencillez de uso, la portabilidad, el soporte multi-lenguaje o sus herramientas.

Gracias al esfuerzo realizado por la comunidad científica, los frameworks de robótica están mejorando rápidamente. A pesar de que la experiencia del usuario es superior en comparación con otros frameworks, se están realizando nuevas mejoras en RoboComp: nuevas características, herramientas, o capacidad de introspección. La instalación automática también está siendo extendida a otros sistemas operativos.

REFERENCIAS

[1] M. Henning y M. Spruiell. *Distributed Programming with Ice*. 2009.

[2] ZeroC. *ZeroC Customers, http://zeroc.com/customers.html*.

[3] M. Montemerlo, N. Roy y S. Thrun. *Perspectives on Standardization in Mobile Robot Programming: The Carnegie Mellon Navigation (CARMEN) Toolkit*. Proc. of International Conference on Intelligent Robots and Systems, 2003.

[4] J. M. Cañas, J. Ruíz-Ayúcar, C. Agüero y F. Martín. *Jde-neoc: component oriented software architecture for robotics*. Journal of Physical Agents, Vol. 1, No. 1, pp 1-6, 2007.

[5] C. Côté, Y. Brosseau, D. Létourneau, C. Raïevsky y F. Michaud. *Using MARIE for Mobile Robot Component Development and Integration*. Software Engineering for Experimental Robotics, Springer, pp. 211-230, 2007.

[6] H. Utz, G. Mayer, U. Kaufmann, y G. Kraetzschmar. *VIP: The Video Image Processing Framework Based on the MIRO Middleware*. Software Engineering for Experimental Robotics, Springer, pp. 325-344, 2007.

[7] P. Newman. *MOOS - Mission Orientated Operating Suite*. Massachusets Institute of Technology, Dept. of Ocean Engineering, 2006.

[8] A. Brooks, T. Kaupp, A. Makarenko, S. Williams y A. Orebäck. *Orca: A Component Model and Repository*. Software Engineering for Experimental Robotics, Springer, pp. 231-251, 2007.

[9] H. Bruyninckx. *Open Robot Control Software: the OROCOS project*. Proc. of International Conference on Intelligent Robots and Systems, pp. 2523-2528, 2001.

[10] B. Gerkey, T. Collet y B. MacDonald. *Player 2.0: Toward a Practical Robot Programming Framework*. Proc. of the Australasian Conf. on Robotics and Automation, 2005.

[11] M. Quigley, B. Gerkey, K. Conley, J. Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler y Andrew Ng. *ROS: an open-source Robot Operating System*. ICRA Workshop on Open Source Software, 2009.

[12] P. Fitzpatrick, G. Metta y L. Natale. *Towards Long-Lived Robot Genes*. Journal of Robotics and Autonomous Systems, vol. 56, num.1, p. 29-45, 2008.