ROS (Remote Optimization Service)

E. Alba, J.G Nieto

Resumen. En este informe se describe el Servicio de Optimización Remota ROS. Este servicio permite a sus clientes el acceso remoto con la posibilidad de ejecutar y manipular algoritmos de optimización basados en técnicas heurísticas, situados en repositorios con recursos hardware y software para optimizar problemas prácticos y reales. Presentamos también la relación con otros trabajos similares, su arquitectura y principales características. Además, se realiza una introducción de las herramientas de comunicación utilizadas, especificación de los tipos de datos con XML y encapsulado de algoritmos a través del servicio ROS. Por último, se muestra una comparativa sobre las distintas formas de configurar este sistema, así como de los resultados obtenidos tras realizar evaluaciones de este servicio sobre distintas configuraciones típicas.

1. Introducción

En la actualidad, están constantemente surgiendo esfuerzos por desarrollar sistemas que integren bibliotecas de algoritmos heurísticos (como por ejemplo algoritmos evolutivos [9]). Además, se intenta ofrecer metodologías y servicios para facilitar la utilización de estos repositorios de software. Una característica común en casi todos estos trabajos es la combinación *XML* [2]*/algoritmos de optimización*, mediante la que se proponen distintas aplicaciones que facilitan tareas como: creación de nuevos algoritmos, compiladores que generan código a partir de especificaciones XML, integración de algoritmos en servicios para su ejecución, e integración XML de objetos y métodos (véase por ejemplo [8][12][13][17]).

En este documento se describe el servicio ROS (*Remote Optimization Service*) [1], mediante el que se pretende proporcionar un servicio de ejecución de algoritmos heurísticos a través de Internet. Con ROS es posible establecer un sistema de servidores que gestionan la inclusión y el funcionamiento de diferentes algoritmos, heterogéneos entre sí, y posiblemente implementados con distintos lenguajes de programación.

Mediante un programa cliente, podemos conectar con estos servidores para interactuar con los algoritmos registrados y activos en ellos. Esta interacción comprende funciones como la selección de algoritmos, introducción de parámetros, ejecución remota y recepción de resultados.

La arquitectura de ROS se ha diseñado para facilitar al administrador del sistema la incorporación de nuevos servidores y algoritmos, creando así un amplio repositorio de algoritmos heurísticos accesibles de forma común y desde clientes distanciados en la red.

Continuando con esta descripción, se expone en la sección 2 un resumen de sistemas existentes con características relacionadas con este servicio. En la sección 3 se define ROS y en la sección 4 se describe su arquitectura, componentes y principales características. En la sección 5 se realiza una breve introducción de la herramientas de comunicación que utiliza ROS, especialmente `WebStream`. La sección 6, describe los *Wrappers* como herramientas para añadir algoritmos a ROS. En la sección 7 se describe la especificación XML `optimization_algorithm`. Por último, en la sección 8 se evalúan las prestaciones reales de ROS con una relación de evaluaciones realizadas sobre este servicio, finalizando en la sección 9 donde se comentan las conclusiones y trabajo futuro a realizar.

2. Estado del Arte

Existen muchos trabajos que intentan cubrir la necesidad de especificar los algoritmos heurísticos mediante XML. El ejemplo más notable es el lenguaje *EAML* [11][12] que especifica algoritmos Evolutivos con etiquetas definidas en su diccionario (DTD [2]) y a partir del cual, se puede generar código C++ mediante un compilador llamado ECC (Evolutionary Computation Compiler). EAML admite evidentes mejoras de diseño para evitar que todos los elementos estén definidos como etiquetas específicas en el diccionario. Además, no ofrece etiquetas de propósito general, de modo que cada vez que se quiera especificar un elemento nuevo hay que actualizar el diccionario. Sin embargo, en este trabajo el objetivo final de XML es la especificación, no la ejecución real y aún menos el paralelismo.

Un sistema que introduce algunas mejoras sobre las limitaciones de EAML es la biblioteca OPEAL [13], que reúne algoritmos evolutivos implementados mediante un lenguaje XML con scripts en el lenguaje Perl llamado `Algorithm::Evolutionary` [13], de modo que ofrece una forma alternativa de representar las estructuras de datos de los algoritmos.

Esta biblioteca se ha ampliado incluyendo la posibilidad de implementar algunos módulos de OPEAL, especialmente los de programación distribuida usando SOAP [14]. En este caso la especificación y la implementación están unidas y es posible una definición limitada de algoritmos básicos. En el wiki *XMLMen* [16] también se exploran los diferentes paradigmas de especificación de algoritmos evolutivos usando XML (inclusive una versión previa de `Algorithm::Evolutionary`), usando finalmente EAML para hacer una serie de experimentos dentro de un sistema denominado Sutherland.

Existen otros servicios disponibles que integran XML con Computación Evolutiva, por ejemplo, el servicio web basado en algoritmos evolutivos *eaWeb* [15], al cual se proporciona una descripción de las funciones a optimizar y los parámetros del algoritmo evolutivo usando XML y devuelve la función optimizada. Este sistema funciona con una idea similar a la que suscita este trabajo, sin embargo, *eaWeb* nos restringe el tipo de algoritmos, estructuras de datos y parámetros a utilizar, los cuales están implementados mediante un conjunto de ficheros XML como parte del propio sistema. Esta característica dificulta la extensibilidad del servicio pues para añadir otro tipo de algoritmo o parámetro es necesario la modificación del sistema y la creación de nuevos ficheros XML.

Otra herramienta disponible en este ámbito es *eaLib* [17]. Se trata de una biblioteca de clases Java mediante la cual un desarrollador con conocimientos en el campo de la computación evolutiva puede crear algoritmos de este tipo de manera sencilla. Además, incluye un método de serialización XML, pero se trata de un método automático de serialización de objetos en Java, no uno específico que sea legible para expertos en algoritmos evolutivos. A diferencia del servicio ROS (objeto de estudio en este documento), el objetivo de *eaLib* no es dar soporte a la ejecución remota de algoritmos heurísticos existentes, sino que está enfocado a la creación de algoritmos evolutivos nuevos.

Dando un paso adelante, se pueden encontrar sistemas que ofrecen una interfaz gráfica de usuario (GUI) para facilitar la creación y posterior ejecución de nuevos algoritmos a partir de componentes agrupados en bibliotecas propias. Por ejemplo, *Evolvica* [19], es el sistema sucesor de *eaLib* (también implementado en Java) y pretende ir más allá pues proporciona un entorno gráfico de desarrollo de algoritmos evolutivos, editor de Java, *debugger* y análisis visual de resultados. *Evolvica* está basado en la plataforma *Eclipse* [20], y usa los fundamentos que provee esta herramienta de trabajo en cuanto a las características Java. Debido a estas características el uso de *Evolvica* está enfocado a usuarios expertos con conocimientos de programación en Java y computación evolutiva. Este sistema se encuentra aún en fase experimental y está sujeto a continuas actualizaciones.

Otro ejemplo es la aplicación *EAVisualizer* [18], también implementada en Java, aunque el usuario no necesita tener conocimientos de Java ni otro lenguaje de programación, pues con este sistema se pueden realizar ejecuciones de algoritmos

evolutivos a partir de la selección de manera interactiva de diferentes componentes predefinidas con anterioridad. Estos componentes implementan las partes de un algoritmo del tipo mencionado (operadores, estructuras de datos, parámetros ,etc). *EAVisualizer* es muy extensible tanto en nuevas componentes evolutivas como en vistas de su interfaz gráfica, que se van incorporando las en sucesivas actualizaciones del sistema.

Sistema/ Característica	Descripción	Modo de Operar	Uso de XML	Lenguaje de Programación	Especificación de Algoritmos	Ofrece GUI
EAML + ECC	Especificación XML y Compilador	Local	Sí	XML y C	Restringida al diseño XML	No
OPEAL	Biblioteca	Local y Distribuida	EAML y SOAP	Perl	Restringida al lenguaje de programación	No
eaWeb	Servicio Web	Web C/S	Sí	ASPX	Restringida al diseño XML	Sí
eaLib	API Java	Local	No	Java	Restringida a las clases del API	No
EAVisualizer	Aplicación (Framework)	Local	No	Java	Restringida a las opciones disponibles	Sí
Evolvica	API y Herramienta de edición	Local	No	Java	Restringida a las clases del API	Sí
ROS	Servicio de Otimización Distribuido	Local y Distribuido	Sí	Cualquiera	Formato E/S de Wrapper (sección 6)	Sí

Tabla 1: Tabla de características de los sistemas relacionados con ROS.

La tabla 1 nos muestra una relación de los sistemas anteriormente citados, destacando una serie de características a partir de las cuales podemos realizar una comparativa: modo de operar, uso de XML, lenguaje de programación, especificación de algoritmos y si ofrece o no GUI. En la última fila, se sitúa el servicio ROS del cual se describen también las características enumeradas anteriormente.

En el caso de los dos primeros ejemplos (EAML+ECC y OPEAL) la característica principal se fundamenta en el uso de XML (en concreto EAML) para la especificación de algoritmos evolutivos. Es decir, implementan completamente el algoritmo mediante EAML para compilarlo y ejecutarlo, ya sea localmente (en el caso de ECC) o de manera distribuida (en el caso de OPEAL).

El objetivo de ROS al respecto es utilizar XML para contener los datos que utiliza el algoritmo (parámetros, resultados, órdenes de ejecución, etc), sin intervenir en la implementación de éste que podrá ser mediante cualquier lenguaje de programación.

El resto de sistemas (*eaWeb*, *eaLib*, *EAVisualizer* y *Evolvica*) ofrecen una funcionalidad parecida. En todos se establecen una serie de funciones, estructuras y tipos de datos a utilizar para especificar algoritmos y ejecutarlos, bien en un servidor web como es el caso de *eaWeb*, o localmente como se realiza en los demás casos. En este aspecto ROS trabaja de manera diferente, pues su misión es incorporar algoritmos ya implementados, y proporcionar mecanismos para la obtención de parámetros y ejecución de manera local o remota. Debido a esta característica, ROS proporciona una mayor libertad en la implementación de los algoritmos con los que puede trabajar.

Además, en todos estos sistemas (excepto *eaWeb*), los algoritmos han de ser implementados en Java, utilizando las clases y/o componentes propias. La implementación de los algoritmos en ROS es totalmente independiente al sistema, es decir, se pueden añadir (mediante un `wrapper`) algoritmos implementados en prácticamente cualquier lenguaje de programación, siempre que cumplan con unos requisitos mínimos de entrada/salida de datos (ver sección 6).

Como última característica, se observa en sistemas como *EAVisualizer* y *Evolvica* al igual que en ROS la capacidad de ofrecer una interfaz gráfica (GUI) para facilitar la utilización del sistema a los usuarios finales. En el caso de *eaWeb*, al ser un servicio Web incorpora formularios (HTML-ASPX) en su aplicación cliente para la obtención y presentación de los datos.

3. ¿Qué es ROS?

ROS *(Remote Optimization Service)* nace con el objetivo de establecer un servicio de optimización en Internet, es decir, facilitar el acceso y uso de múltiples algoritmos (especialmente heurísticos) creados por diferentes desarrolladores, cubriendo así la necesidad de agrupar algoritmos muy heterogéneos e implementados con diferentes lenguajes de programación, en un sistema al cual pueden acceder de manera sencilla multitud de usuarios distanciados en la red.

De este modo, se establecen dos tipos de actores principales que actúan sobre ROS: el *desarrollador*, que añade su algoritmo en un servidor (Worker) mediante un *Wrapper* (sección 4), y el *usuario*, que utilizando el programa Cliente de ROS puede interactuar de manera remota con los algoritmos disponibles y así resolver un problema (típicamente de optimización). Además, es posible configurar ROS para que trabaje de manera local o bien integrado en una LAN, estableciendo así una plataforma de prueba y ejecución de algoritmos.

4. Descripción de ROS

Debido a su carácter distribuido y multiplataforma, ROS está implementado con JAVA [6] y utiliza herramientas de comunicación (sección 5) para establecer una jerarquía de servidores mediante el modelo Cliente/Servidor (C/S).

ROS se organiza conectando cuatro tipos de componentes: el cliente, el servidor primario, el servidor de distribución y los servidores de proceso o *Workers*. En la figura 1 podemos ver un esquema de la arquitectura de ROS.

1. El cliente ROS (*Client*) es una aplicación que ofrece una interfaz gráfica a través de la cual un usuario puede realizar una serie de funciones sobre el sistema. Tales funciones son: la identificación o registro de usuario en el sistema, selección de tipo de algoritmo, selección de servidores disponibles (en función del tipo de algoritmo), y ejecución remota del algoritmo seleccionado. Internamente, el cliente ROS establece la comunicación con los demás componentes del sistema (servidor primario y servidor de distribución) para llevar a cabo el intercambio de información necesaria en cada función.

2. El servidor primario (*Primary Server*) proporciona al cliente el servicio de identificación en el sistema (login y password), además del servicio de información sobre los tipos de algoritmos y direcciones de los servidores accesibles. Es el primer servicio al cual accede un cliente cuando comienza una ejecución de ROS.

3. Con el servidor de distribución (*Server*) se redireccionan todos los flujos de información existentes entre los clientes conectados y los servidores de proceso. El *Server* es así una pieza fundamental para lograr tanto la escalabilidad como la distribución del sistema en red, pues es el organizador de la comunicación entre el cliente y el algoritmo. Es decir, actúa como un servidor Proxy respecto a la interacción entre el cliente y el servidor de proceso. De este modo, la incorporación de un nuevo servidor de proceso o algoritmo en el sistema pasa por su registro en el *Server*. Este registro comprende la introducción de una nueva entrada en un fichero de configuración de los tipos de algoritmos y direcciones de servidores de proceso.

4. El servidor de proceso (*Worker*) aloja los algoritmos en el sistema y los manipula según las órdenes especificadas desde el cliente. La forma de alojar los algoritmos es mediante los *Wrappers* que encapsulan el código de estos algoritmos y ofrecen los métodos necesarios para su manipulación desde el exterior. Su función principal es ofrecer servicio de ejecución de los algoritmos que contiene, cada vez que reciba una petición adecuada desde un cliente. La petición de ejecución, en realidad no se realiza directamente desde el cliente, si no que se lleva a cabo a través de servidor de distribución (anteriormente comentado).

Una vez atendida una petición y realizada la ejecución de un algoritmo seleccionado, el servidor de proceso devolverá una respuesta por el mismo cauce al servidor de distribución.

A partir de la siguiente ilustración (Figura 1), podemos observar un esquema resumido de la arquitectura de ROS en una distribución hipotética de sus componentes y la interrelación que existe entre ellos.

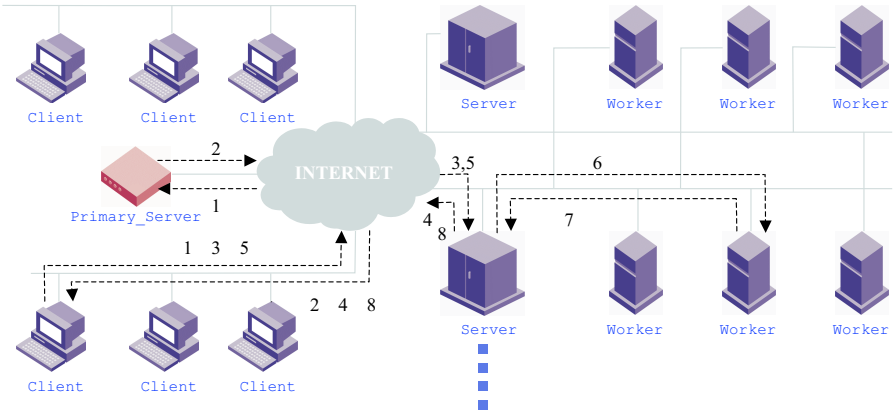

Figura 1: Esquema resumido de la arquitectura de ROS.

Generalmente, el sistema se compone de una serie de subredes donde se sitúan las diferentes partes de ROS. En la parte izquierda se disponen las subredes que contienen los clientes (*Client*) y una subred en la cual se encuentra el servidor primario (*Primary Server*). En el centro están los servidores de distribución (*Server*) y a partir de ellos se organizan las subredes con los servidores de proceso (*Worker*). Todas las subredes se comunican a través de Internet. Este es un esquema ideal, en la realidad se pueden disponer distintos componentes en una misma subred e incluso es posible configurar ROS para que trabaje de manera local con todos los componentes en una misma máquina.

Para seguir un ejemplo de funcionamiento completo nos fijamos en las líneas discontinuas que aparecen en la figura, pues nos dan una idea de la secuencia y dirección del flujo de datos a través del sistema en un ciclo de ejecución. A continuación enumeramos los pasos elementales del funcionamiento de ROS.

1. *Identificación de usuario*. El usuario introduce su login y password en el formulario inicial del cliente y conecta con el servidor primario para su comprobación o registro. Una vez identificado el usuario, el servidor primario le envía de vuelta al cliente un mensaje con la aceptación o rechazo del usuario según corresponda. En caso de ser un nuevo registro, el servidor primario crea una nueva entrada en su fichero de usuarios y envía al cliente la aceptación.

Tras una identificación con éxito, el cliente pasa a la fase de selección de tipo de algoritmo y servidor de distribución.

2. *Selección del tipo de algoritmo, dirección de servidor y lenguaje de programación.* En esta fase, el cliente realiza una petición de tipos de algoritmos al servidor primario, el cual le envía de vuelta una relación clasificada de los tipos de algoritmos disponibles actualmente, las direcciones de los servidores de distribución a los cuales se deben acceder, y por último los lenguajes de programación en los que están implementados los algoritmos.

3. *Petición de entorno de algoritmo.* El cliente, en base al tipo de algoritmo, servidor y lenguaje de programación seleccionado en el punto anterior, realiza una petición al servidor de distribución. Con esto se consigue establecer el entorno del algoritmo seleccionado, es decir, los tipos de ficheros XML y la ventana gráfica específica del tipo de algoritmo.

4. *Obtención del entorno de algoritmo.* El servidor de distribución genera por primera vez el fichero XML adecuado al algoritmo seleccionado y se lo envía al cliente. El cliente, al recibir este fichero, genera automáticamente la ventana gráfica adecuada al formato del fichero XML.

5. *Introducción de parámetros de entrada.* En esta fase, el cliente dispone de los campos de parámetros de entrada del algoritmo, así que el usuario ya puede introducir estos parámetros e iniciar una ejecución remota realizando la petición al servidor de distribución.

6. *Ejecución del algoritmo.* El servidor de distribución recibe la petición de ejecución desde el cliente. Obtiene el tipo de algoritmo a ejecutar, y en base a éste selecciona la dirección del servidor de proceso al cual realizar la petición. Una vez conectado con este servidor (*Worker*), le envía los datos obtenidos desde el cliente para que se inicie la ejecución remota.

7. *Obtención de resultados.* Al terminar la ejecución del algoritmo se escriben los resultados en el fichero XML y se devuelve de nuevo al servidor de distribución.

8. *Lectura de Resultados.* El servidor de distribución finalmente devuelve el fichero XML con los resultados de la ejecución al cliente, el cual se encargará de mostrar los resultados por pantalla o generar mensajes de error en otro caso.

Cabe destacar el papel de XML dentro del flujo de información que circula entre los componentes de ROS. También es importante que profundicemos en los *Wrappers* pues son pieza fundamental en el funcionamiento de este servicio.

5. El Intercambio de Información en ROS

Debido a su naturaleza distribuida, ROS requiere de un sistema para la conexión e intercambio de información entre sus componentes que generalmente estarán instalados en máquinas remotas. Se disponen de dos versiones diferentes de ROS dependiendo del sistema de comunicación que se utilize:

- ROS con intercambio de información mediante RPC con SOAP [4].
- ROS con intercambio de información mediante `WebStream`.

En la versión SOAP de ROS, se establece el intercambio de información mediante llamadas a procedimientos remotos entre las máquinas C/S. Se crean objetos remotos en los servidores accesibles desde el cliente, y a través de estos objetos se realiza el intercambio de información. En la siguiente figura se muestra la estructura de servicios SOAP que se establece en ROS.

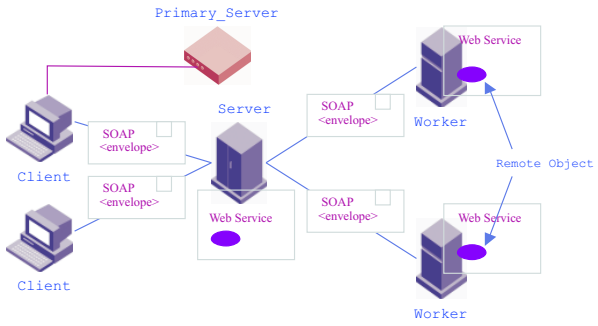

Figura 2: Esquema resumido de la arquitectura SOAP de ROS.

En este ejemplo muy resumido (Figura 2), ROS/SOAP consta de los servidores de proceso (*Workers*). Estos se implementan mediante servicios web (Apache [5]), donde está registrado el objeto remoto que realiza la función del servidor de proceso. Cada objeto remoto es accesible por el servidor de distribución, que a su vez proporciona otro objeto remoto (que realiza la función del servidor de distribución) mediante un servicio web al cliente. Así se establece el grafo de comunicaciones de ROS mediante secuencias C/S hasta "conectar" el algoritmo (situado en un *Worker*) con el cliente.

Esta implementación de ROS usando SOAP tiene la ventaja de hacer el sistema muy escalable y ofrece un medio muy estandarizado (XML envelope) para una posible adhesión con otros componentes o aplicaciones que utilicen el mismo estándar. Por otra parte, el hecho de tener que instalar servidores Web adicionales para el registro de los objetos remotos supone una complejidad añadida.

Además, el sistema debe interpretar el protocolo RPC de SOAP con lo que se incrementa la carga de proceso a realizar por el servicio.

Para los casos en que se requiera una menor carga de proceso, o sea complicado la instalación de servidores web adicionales, existe una segunda versión que utiliza `WebStream` [7].

`WebStream` (`WS`) es una herramienta para la comunicación entre procesos (C/S) mediante *Paso Síncrono de Mensajes* y con la que es posible establecer un servicio *Orientado a la Conexión*. El objetivo de `WebStream` es ofrecer una serie de mejoras sobre otros sistemas de comunicación de procesos remotos (sockets, RPC, RMI, etc), para facilitar al programador la implementación de sistemas C/S con la posibilidad de intercambiar tipos de datos construidos como arrays, ficheros e incluso objetos genéricos, sin que sean necesarios sistemas pesados computacionalmente o el uso de XML.

`WebStream` está implementado mediante una clase Java que encapsula el servicio de `sockets TCP` de la biblioteca `java.net`, de modo que proporciona una interfaz de métodos muy parecida a la de `sockets` como métodos para el establecimiento de una conexión, envío y recepción de mensajes, finalización de la conexión, etc. Este servicio se presta, por defecto en el puerto 80 por lo que las usuales trabas de seguridad no serán ahora un problema.

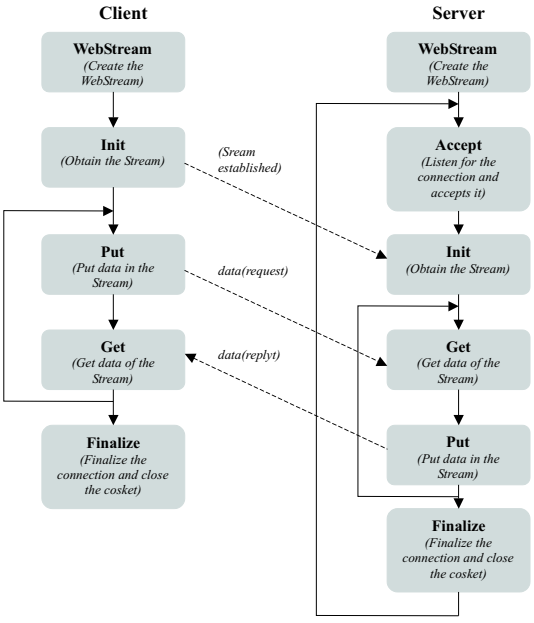

Figura 3: Funcionamiento de WebStream (C/S).

Para implementar un sistema de comunicación entre procesos (C/S) mediante `WebStream` se deben crear instancias de los constructores y se invocan los métodos para el establecimiento de la conexión como muestra la figura 3.

`WebStream` proporciona la posibilidad de intercambio de información *full-duplex* por lo que se realizan escrituras y lecturas en ambos sentidos de la conexión. Aunque un aspecto a considerar es el tipo de los datos que se leen y escriben, pues para que no haya errores en la recepción, el método que recibe un determinado tipo de datos desde el `stream` debe ser correspondiente al método que envió estos datos, por ejemplo: si hacemos `put(float [] p)` [7] en el cliente, se debe hacer el correspondiente `get(float [] p')` en el servidor, o bien los resultados pueden ser inesperados.

ROS utiliza `WebStream` para establecer conexiones e intercambio de información entre los componentes que lo integran, de manera que implementa el cliente como cliente `WS`, el servidor primario y servidor de proceso como servidores `WS` y el servidor de distribución se implementa como servidor `WS` respecto al cliente ROS y como cliente `WS` respecto a los servidores de proceso.

6. Wrappers

Mediante los Wrappers se encapsulan los algoritmos que se pretenden añadir en el sistema, concretamente en los servidores de proceso (*Workers*). Estos algoritmos pueden ser muy heterogéneos en cuanto al lenguaje de programación (C, C++, Java, Modula 2, etc) e incluso el paradigma utilizado para su desarrollo (Orientado a Objetos, Modular, Imperativo, etc).

Debido a esta gran variedad de tipos debemos programar una clase `Wrapper\_` (Java) específica para cada algoritmo, aunque todas estas deben heredar de la clase `Wrapper` padre (ver Figura 4) pues ofrecen los mismos métodos y es instanciada del mismo modo (independientemente del tipo de algoritmo).

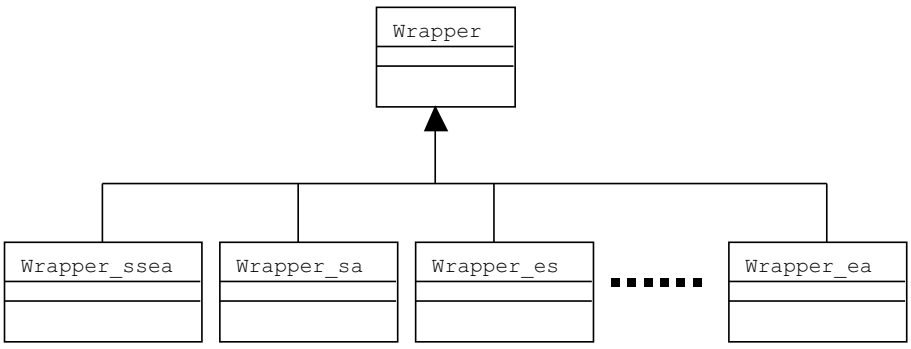

Figura 4: Diseño simplificado del diagrama de clases de Wrapper.

Cada *Wrapper* ofrece en su interfaz el método público `run()` que a su vez llama a los demás métodos dedicados a hacer el "tunneling" del algoritmo:

- `private void getConfigurationFile(String xml_file_name, String config_file_name)`. Genera el fichero de configuración adecuado para el algoritmo a partir del fichero XML (ver sección 7) con los parámetros.
- `private void compile()`. Ejecuta las órdenes de compilación adecuadas al código del algoritmo.
- `private void execute()`. Ejecuta las órdenes de ejecución adecuadas al código del algoritmo.
- `private void getXmlResult(String results_file_name, String xml_file_name)`. Inserta en el fichero XML los resultados de la ejecución a partir del fichero de resultados del algoritmo.

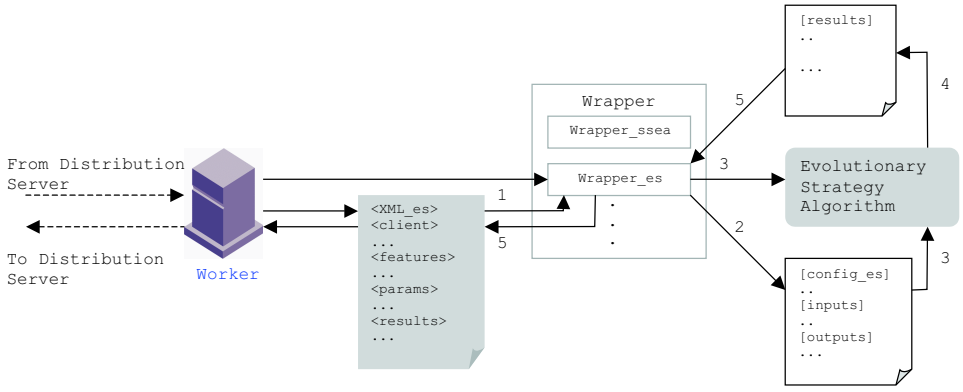

Figura 5: Relación de un Wrapper respecto al algoritmo que encapsula.

Para comprender el funcionamiento de los *Wrappers* nos basamos en un ejemplo típico de uso (ilustrado en la figura 5), puesto que su comportamiento es similar en todos los casos. En concreto nos fijamos en un Wrapper para un algoritmo de *Estrategia Evolutiva* (Evolutionary Strategy) [26] al que llamamos `Wrapper_es`:

1. El servidor de proceso (*Worker*) obtiene el fichero XML (enviado por el servidor de distribución) y lo hace disponible al `Wrapper_es`.
2. A partir del fichero XML con los parámetros de entrada, el `wrapper` genera el fichero de configuración específico al algoritmo (método `getConfigurationfile`).
3. El `Wrapper_es` extrae la orden de compilación (si es necesario) y ejecución del algoritmo, lanza dicha orden de compilación/ejecución y espera a que termine este proceso (métodos `compile` y `execute`). El algoritmo internamente obtiene los parámetros de su fichero de configuración específico generado en el paso anterior.

4. Cuando finaliza la ejecución del algoritmo, escribe los resultados en el fichero correspondiente del propio del algoritmo.

5. El `wrapper_es` añade los resultados (obtenidos del fichero de resultados) en el fichero XML. Tras esto, el servidor de proceso dispone del fichero XML completo con parámetros y resultados, así que lo envía de vuelta al servidor de distribución.

Debido al modo de proceder automático de los `wrappers`, los algoritmos que se pretendan añadir al sistema deben cumplir la restricción de entrada/salida de datos impuesta en este esquema. Es decir, los datos de entrada se proporcionan al algoritmo mediante un fichero (en la figura anterior `[config_es]`), y los datos de salida (resultados) los escribirá el algoritmo en otro fichero diferente (en la figura `[results]`).

7. XML en ROS

Para el servicio ROS no es necesario que la especificación XML describa completamente el algoritmo que se pretende manipular. En este sistema, se utiliza XML para especificar la configuración de un algoritmo, de modo que la información que contiene es del tipo: parámetros de entrada, parámetros de salida, información del cliente (usuario), características sobre la implementación del algoritmo, lenguaje de programación, etc. En la figura 6 se muestra un documento XML típico de ROS para la configuración de un algoritmo de *Estrategia Evolutiva* continuando con el ejemplo de la sección anterior.

En este sentido, se ha definido el diccionario (*DTD - Data Type Document*) `optimization_algorithm.dtd` [21][22] creado para este servicio, a partir del cual se especifican todos los ficheros XML que utiliza ROS. Cada fichero XML se organiza jerárquicamente de manera que hay un elemento raíz `<optimization_algorithm>` que contiene a los cuatro elementos principales:

- `<client>`, contiene elementos referidos a los datos del cliente (IP, ID, etc).
- `<features>`, sus elementos describen las características del algoritmo (lenguaje, órdenes, etc).
- `<params>`, elementos que especifican los parámetros de entrada.
- `<results>`, elementos que especifican los parámetros de salida (resultados).

Además, cuenta con elementos de propósito general llamados elementos `<others\_...>` con los que podemos especificar otros tipos de datos no contemplados en el DTD.

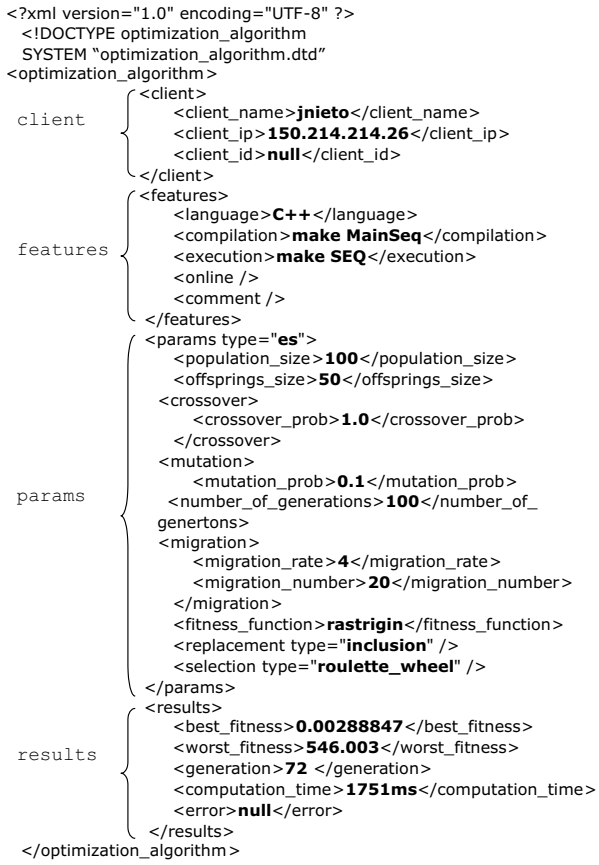

Figura 6: Documento XML de un algoritmo de *Estrategia Evolutiva*.

El servicio ROS utiliza este formato de ficheros XML para contener los datos que se intercambian entre sus distintos componentes. Inicialmente, el fichero XML es generado en el servidor de distribución de acuerdo con el tipo de algoritmo seleccionado por el usuario en el cliente. En esta fase se introducen los datos de cliente y características del algoritmo seleccionado en las etiquetas `<client>` y `<features>` respectivamente, además del tipo de algoritmo en el atributo `type` de la etiqueta `<params type=‘‘tipo de algoritmo’’>`.

De este modo el fichero es enviado al cliente donde se introducen los parámetros de entrada del algoritmo, cada uno en la sub-etiqueta correspondiente dentro de `<params ...>`. Cuando el cliente inicia la ejecución, envía el fichero de vuelta al servidor de distribución para que éste los envíe al servidor de proceso correspondiente.

Cuando el fichero XML llega al servidor de distribución, el `wrapper` se encarga de obtener los datos e introducir los resultados en la etiqueta `<results>`, cada uno dentro de su sub-etiqueta correspondiente.

Al finalizar, el servidor de proceso devuelve el fichero XML completo por el mismo cauce hacia el cliente, el cual muestra los resultados al usuario. El proceso es el descrito en la sección anterior.

Esta especificación XML sobre algoritmos evolutivos busca sencillez y facilidad de uso pero abarcando todo el conjunto de argumentos necesarios en esta materia.

8. Evaluación de ROS

Para la realización de una evaluación del servicio ROS se han considerado dos tipos de configuraciones de red: a través de una LAN y localmente, trabajando en cada caso sobre un conjunto común de algoritmos:

- En la configuración LAN, se ha utilizado una red *Ethernet* a 100Mbps, los servidores se ejecutaron en máquinas con procesadores *Intel* 2,4 GHz y memoria RAM de 512MB y el cliente se ejecutó en una máquina con procesador *Intel* 2,6 GHz y memoria RAM de 512MB.

  En cuanto a los Sistemas Operativos utilizados, el servidor de distribución y servidor primario se ejecutaron sobre *Linux Suse*, el cliente sobre *Windows* y los servidores de distribución se ejecutaron sobre distintos Sistemas Operativos (*Linux/Windows*) dependiendo de los requisitos de los algoritmos que manejan.

- Mediante Configuración Local, se ejecutaron todos los servidores y el cliente en una misma máquina con procesador *Intel* 2,6 GHz y memoria RAM de 512MB. En este caso, se realizaron las pruebas con todos los componentes de ROS sobre *Windows* y sobre *Linux*, dependiendo de los requisitos de los algoritmos.

Los algoritmos ejecutados mediante ROS en estas evaluaciones son de naturaleza muy diversa e implementados en diferentes lenguajes de programación, los cuales se describen a continuación:

- *Algoritmo Evolutivo de Estado Estacionario* [23]. Implementado en *Java* para sistema operativo *Windows/Linux*. Algoritmo Genético [9] con la principal característica de que son reemplazados en cada generación muy pocos individuos (típicamente uno o dos). Resuelve el problema *OneMax* [29].

- *Algoritmo Evolutivo Celular* [24]. Implementado en *Modula 2* para sistema operativo *Windows*. Algoritmo genético que usa una disposición espacial para los individuos en una simple población. Estos individuos están fuertemente conectados. En este caso se realiza un proceso de *Estado Estacionario* sobre cada población en paralelo. Resuelve el problema *Rastrigin* [30].

- *Algoritmo Evolutivo Distribuido*[25]. Implementado en *Modula 2* para sistema operativo *Windows*. Algoritmo genético compuesto de sub-algoritmos distribuidos con conexión (más devil que en el caso celular) entre ellos y que manejan varias sub-poblaciones (>>1) cada uno. El algoritmo utilizado se compone de varios sub-algoritmos de *Estado Estacionario* distribuidos. Resuelve el problema *Rastrigin*.

- *Estrategia Evolutiva* [26]. Implementado en *C++* para sistema operativo *Linux*. Algoritmo evolutivo que trabaja con una población de individuos que pertenecen al dominio de los números reales, y que mediante los procesos de mutación y de recombinación evolucionan para alcanzar el óptimo de la función objetivo. Resuelve el problema *Rastrigin*.

- *Recocido Simulado* [27]. Implementado en *C++* para sistema operativo *Linux*. El *Recocido Simulado* es uno de los algoritmos Meta-heurísticos clásicos, y se basa en una búsqueda local que permite movimientos ascendentes para evitar quedar atrapado prematuramente en un óptimo local, utilizando para esto una función de temperatura. Resuelve el problema *OneMax*.

- *Algoritmo Genético (Mareas)* [28]. Implementado en *C++* para sistema operativo *Windows*. Algoritmo genético con selección basada en *Estado Estacionario* para la predicción de series temporales. Debido a su naturaleza, este algoritmo utiliza una gran cantidad de datos en su proceso. Resuelve el problema de *Predicción de Series Temporales*.

La evaluación de estos algoritmos depende de la complejidad de los problemas que resuelven. Concretamente se utilizan los tres tipos de problemas siguientes:

- *OneMax* (o *BitCounting*). Consiste en maximizar el número de *unos* de una cadena de bits. El objetivo es encontrar una cadena de bits $\vec{x} = \{x_1, x_2, \ldots, x_N\}$, con $x_i \in \{0, 1\}$, que maximize la siguiente ecuación:

$$F(\vec{x}) = \sum_{i=1}^{N} x_i \tag{1}$$

- *Rastrigin*. Consiste en encontrar el mínimo global dentro de un espacio de búsqueda con un gran número de mínimos locales. Debido a las dimensiones del espacio de búsqueda y al gran número de mínimos locales es una función de gran complejidad ($\Theta(n \cdot \ln(n))$ donde n es la dimensión del problema).

$$F(\vec{x}) = A \cdot n + \sum_{i=1}^{n} x_i^2 - A \cdot \cos(\omega \cdot x_i) \tag{2}$$
$$A = 10 \ ; \ \omega = 2 \cdot \pi \ ; \ x_i \in [-5{,}12, 5{,}12]$$

La evaluación de esta función viene determinada por las variables externas A y w, las cuales controlan la amplitud y frecuencia de modulación respectivamente.

- *Predicción de Series Temporales.* El objetivo es que dada una muestra consecutiva de horas, predecir la evolución de una marea. Si se introducen una muestra de valores de 24 horas consecutivas, predecir los valores de la hora siguiente (25). Los individuos serán vectores de 50 elementos:

  $$\vec{x} = \{(c\_sup1, c\_inf1, \ldots, c\_sup24, c\_inf24, prediccin, error)\}$$

  Este patrón nos indica que para una muestra de 24 horas consecutivas *(h1, ..., h24)* si para todo i se cumple que *c_ infi <hi<c_ supi* entonces la hora siguiente será *predicción* con un error aproximado de *error*.

En las tablas 2 y 3, se muestran los resultados respecto al *tiempo de ejecución* de los algoritmos y *tiempo de comunicación* desde que el cliente inicia la orden de ejecución hasta que recibe los resultados. Los valores están calculados en función de la media aritmética y la desviación típica de un total de cincuenta evaluaciones en cada caso. [1]

Como puede observarse en estas tablas de evaluaciones, las medias de los tiempos de ejecución (media t.e) en la configuración Local de ROS son generalmente más bajas que en la configuración en LAN del sistema, debido a la diferencia de potencia del procesador donde se ejecutan los algoritmos. No obstante, hay una excepción en el tiempo de ejecución del algoritmo genético *Mareas*, el cual requiere una mayor cantidad de recursos y su ejecución en la configuración Local debe coexistir con los demás procesos de ROS (cliente, servidor de distribución, servidor primario y servidor de proceso). Todo esto implica que para este tipo concreto de algoritmo, el tiempo de ejecución en la configuración local es mayor al de la configuración LAN, en la cual el algoritmo se ejecuta en un servidor remoto sin más carga de proceso.

Desde el punto de vista de la media del tiempo de comunicación (media t.c) se observa una mayor diferencia. Para los tres primeros tipos de algoritmos evaluados en la tablas, el tiempo de comunicación requerido en la configuración LAN es mayor que en la configuración Local de ROS. Esto es lógico debido a que en la configuración LAN la comunicación se efectúa entre procesos situados en puntos distanciados de una red, mientras que en la configuración Local todos los procesos están en la misma máquina.

[1] media t.e (media del tiempo de ejecución),media t.c (media del tiempo de comunicación),des-tip t.e (desviación típica del tiempo de ejecución), des-tip t.c (desviación típica del tiempo de comunicación), Len Prog (lenguaje de programación).

Algoritmo/ Características	S.O	Len Prog	media t.e	des-tip t.e	media t.c	des-tip t.c
Algoritmo de Estado Estacionario	Win Linux	Java	296.2800 ms	68.9493 ms	1.51e+003 ms	549.919 ms
Algoritmo Evolutivo Celular	Win	Modula 2	50.4200 ms	13.2574 ms	446.4000 ms	294.8466 ms
Algoritmo Evolutivo Distribuido	Win	Modula 2	2.08e+003 ms	129.1549 ms	370.8800 ms	147.1280 ms
Estrategia Evolutiva	Linux	C++	3.44e+003 ms	92.8843 ms	52.8476 ms	256.076
Recocido Simulado	Linux	C++	126 ms	43.5328 ms	264.6400 ms	28.9191 ms
Algoritmo Genético (Mareas)	Win	C++	1.48e+005 ms	699.41 ms	652.42 ms	139.70 ms

Tabla 2: Tabla de evaluaciones de ROS en una configuración LAN.

Algoritmo/ Características	S.O	Len Prog	media t.e	des-tip t.e	media t.c	des-tip t.c
Algoritmo de Estado Estacionario	Win Linux	Java	201.760 ms	18.0832 ms	995.960 ms	61.3501 ms
Algoritmo Evolutivo Celular	Win	Modula 2	139.840 ms	40.1763 ms	339.020 ms	40.298 ms
Algoritmo Evolutivo Distribuido	Win	Modula 2	3.63e+003 ms	61.2494 ms	355.10 ms	83.912 ms
Estrategia Evolutiva	Linux	C++	2 ms	0 ms	2.53e+003 ms	48.1349
Recocido Simulado	Linux	C++	1 ms	0 ms	1.45e+003 ms	42.1422 ms
Algoritmo Genético (Mareas)	Win	C++	1.98e+005 ms	1.55e +004 ms	1.05e+003 ms	729.704 ms

Tabla 3: Tabla de evaluaciones de ROS en una configuración local.

Sin embargo, los tres últimos algoritmos (Estrategia Evolutiva, Recocido Simulado y A.G Mareas) presentan un tiempo de comunicación en configuración local mayor que en configuración LAN. Posiblemente esto se debe a que el tiempo de comunicación engloba preproceso de análisis XML y escritura en ficheros, lo cual puede incrementar el tiempo de comunicación en la configuración local pues requiere una mayor carga de proceso.

Las siguientes gráficas (Figura 7 y 8) ilustran la progresión del tiempo de computación (en configuración LAN y Local respectivamente) resultado de las 50 evaluaciones de los algoritmos descritos.

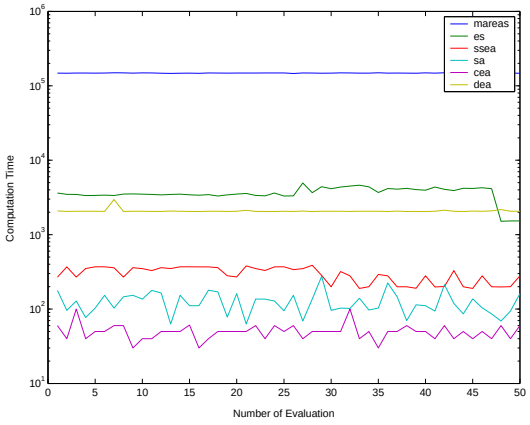

Figura 7: Gráfica de evaluaciones: tiempo de computación en configuración LAN de ROS.

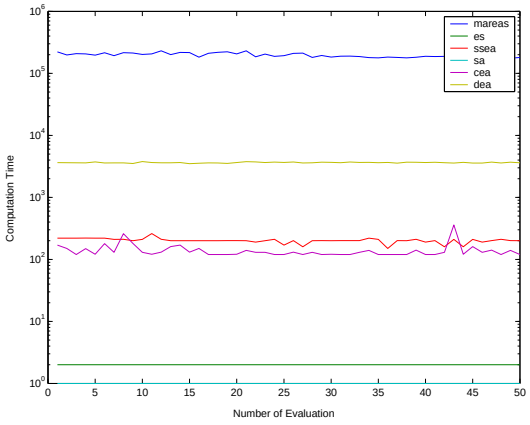

Figura 8: Gráfica de evaluaciones: tiempo de computación en configuración Local de ROS.

Además de lo comentado anteriormente sobre el tiempo de ejecución, se observa en estas gráficas una mayor dispersión en los tiempos de ejecución en la configuración LAN. Esto es debido a que los algoritmos ejecutados deben coexitir con otras aplicaciones independientes de ROS, al contrario que en la configuración Local en la que solamente están en ejecución los procesos de ROS.

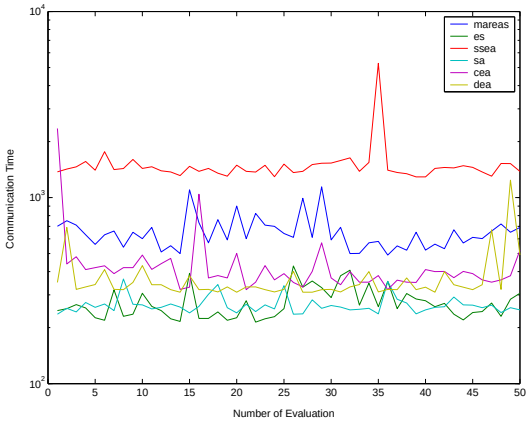

Figura 9: Gráfica de evaluaciones: tiempo de comunicación en configuración LAN de ROS.

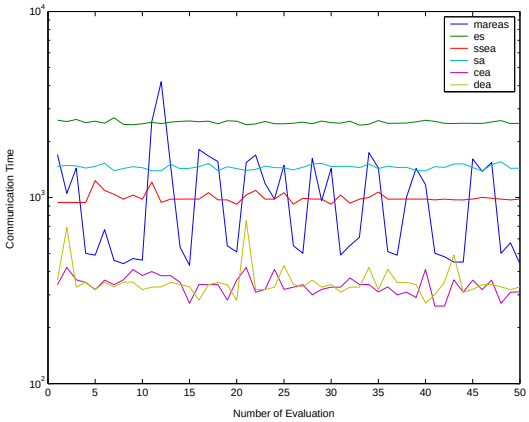

Figura 10: Gráfica de evaluaciones: tiempo de comunicación en configuración Local de ROS.

Las gráficas de las figuras 9 y 10 muestran el resultado de las evaluaciones de los algoritmos respecto al tiempo de comunicación. En este aspecto, las gráficas muestran una mayor dispersión en los resultados de la configuración LAN, pues la comunicación entre procesos está sujeta a las condiciones de tráfico y congestión de la red.

Uno de los aspectos más importantes para estudiar el comportamiento de ROS en una red es medir el flujo de datos que surge en el intercambio de información entre los diferentes componentes del sistema.

De esta forma, podemos ver la interacción entre varios procesos de ROS en cuanto al tráfico que generan en la red, además de su coexistencia con otras aplicaciones que hacen uso de las mismas máquinas y canales de comunicación.

En las figuras 11 y 12, se muestran diagramas (generados mediante la aplicación *Etherape* [31]) que expresan el tráfico y la dirección de los datos que intercambian los procesos de ROS en una evaluación típica a través de una LAN.

Disponemos los servidores en una LAN con todos los servidores de proceso en máquinas internas, el servidor primario y el servidor de distribución están en una máquina accesible desde el exterior pues es la que da servicio a los clientes. Las direcciones de las máquinas donde se ejecutan los procesos son:

- *Cliente* ROS: `c24.lcc.uma.es`.
- *Servidor Primario*: `mallba11.lcc.uma.es`.
- *Servidor de Distribución*: `mallba11.lcc.uma.es` (dirección externa) y `192.168.0.17` (dirección interna).
- *Servidor de Proceso*: `192.168.0.13` (Linux) y `192.168.0.16` (Windows).

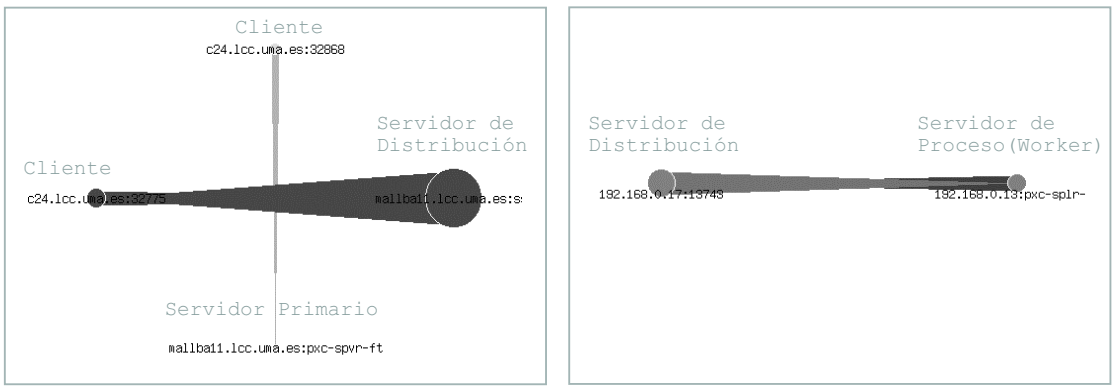

Figura 11: Tráfico generado en una ejecución de ROS respecto al protocolo TCP.

En la figura 11 de la izquierda, se puede ver el tráfico generado por el intercambio de información que se realiza entre el cliente y el servidor primario (estela vertical) y entre cliente y el servidor de distribución (estela horizontal).

El trazo más grueso corresponde a la comunicación que se realiza entre el servidor de distribución y el cliente, concretamente se trata del envío de un fichero XML generado inicialmente en el servidor de distribución. En cambio, la comunicación entre el cliente y el servidor primario desprende el trazo más fino (vertical), puesto que esta transacción corresponde a identificación de usuarios y selección de algoritmos que requiere un menor volumen de datos.

En la figura 11 de la derecha, se expresa el flujo entre el servidor de distribución (una vez realizada la petición del cliente) y el servidor de proceso seleccionado, el cual ejecuta el algoritmo y devuelve los resultados en sentido inverso. En este caso el grosor de la estela es muy parecido en ambos sentidos pues se intercambia prácticamente la misma cantidad de datos mediante un fichero XML.

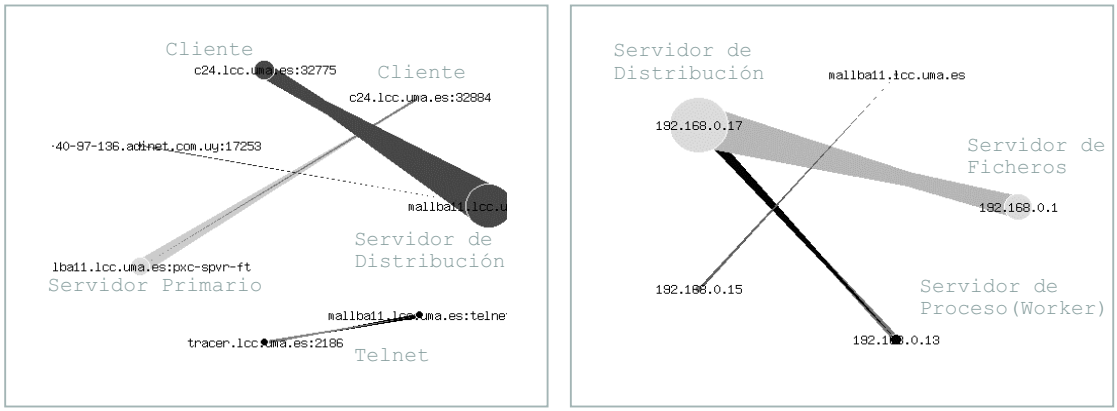

Figura 12: Tráfico generado en una ejecución de ROS respecto al protocolo IP.

En esta representación se ha tenido en cuenta la comunicación respecto al protocolo TCP. En la figura 12 se realiza la misma operación representando la gráfica respecto al protocolo de red IP. Ahora se puede observar la aparición de tráfico generado por otras aplicaciones que coexisten en el mismo entorno que ROS.

La figura 12 de la izquierda es similar a su homóloga en la figura 11, aunque aparece otro flujo independiente a ROS causado por una ejecución de Telnet. En la figura 12 derecha el servidor de distribución (con IP interna `192.168.0.17`) realiza la petición al servidor de proceso situado en `192.168.0.13`, además de una lectura de `192.168.0.1` que es el servidor de ficheros (independiente a ROS) de la LAN.

Aunque en este ejemplo la convivencia con otras aplicaciones que utilizan la red no ejercen prácticamente influencia sobre el funcionamiento de ROS, en otras condiciones en las que existan otras aplicaciones que requieran la transacción de un mayor volumen de datos (por ejemplo FTP), pueden influir de manera negativa en la velocidad de comunicación de este servicio.

Por último, cabe señalar que además de configurar ROS para que trabaje localmente o distribuido en una LAN, también es posible configurar este servicio a través de una red WAN e incluso en Internet. Simplemente hay que instalar los distintos componentes en máquinas situadas en redes remotas (tal y como muestra la figura 13), e introducir las entradas correspondientes a las direcciones de máquinas y tipos de algoritmos en los ficheros de configuración.

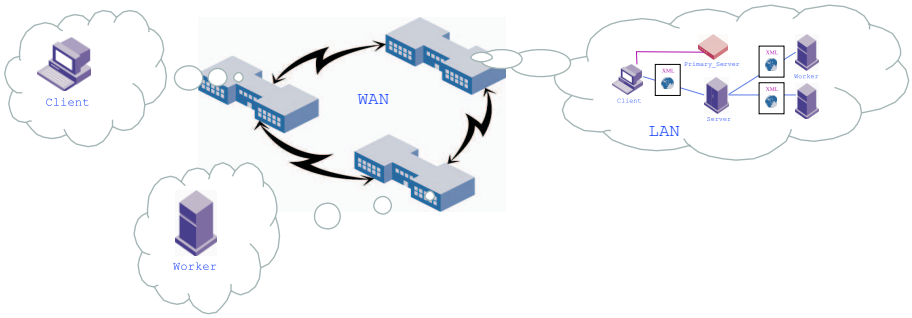

Figura 13: Configuración LAN/WAN de ROS.

Se puede obtener el software necesario para utilizar ROS en el sitio web[1] dedicado, además es posible hacer pruebas conectando con los servidores activos cuyas direcciones están configuradas en el mismo servicio.

9. Conclusiones y Trabajo Futuro

En este documento se ha presentado una descripción muy general de ROS, un servicio que proporciona el acceso y ejecución remota de algoritmos heurísticos a través de Internet. Se basa en una arquitectura cliente/servidor ligado a la especificación XML `optimization_algorithm` desarrollada para el propio sistema. Además, ofrece funcionalidades que facilitan la utilización del servicio tanto para los usuarios (mediante la interfaz gráfica en el cliente), como para los desarrolladores de algoritmos (utilizando la herramienta de *Wrappers*).

Debido a la gran variedad de algoritmos que se pretenden añadir al servicio, y el gran número de argumentos diferentes que maneja cada uno, es necesario generalizar la especificación XML y con esto la funcionalidad del servicio. Esto se refleja en la necesidad de generar un tipo de fichero XML y una subclase `Wrapper` para cada tipo algoritmo, con lo que se corre el riesgo de hacer crecer el sistema complicando su mantenimiento. En este sentido, se pretende realizar en las futuras fases de este proyecto métodos para agilizar la incorporación de los nuevos algoritmos al sistema, imponiendo restricciones en el desarrollo de estos algoritmos, implementando clases que manipulen grupos de algoritmos e incorporando nuevas interfaces gráficas para facilitar su uso a desarrolladores y usuarios.

Referencias

[1] ROS, Remote Optimization Service. Disponible en `http://tracer.lcc.uma.es/ros/index.html`.

[2] E R Harold, XML Bible, IDG Books, 1999.

[3] B McLaughlin, Java and XML, O'Relly, 2001.

[4] R Englander, Java and SOAP, O'Reilly, 2002.

[5] Sitio Web de Apache Web Server. Disponible en `http://www.apache.org`.

[6] A Pew, Instant Java, The Sunsoft Press - Pretince Hall, 1996.

[7] J.G. Nieto, La Clase WebStream, March 2005.

[8] J.J. Merelo, J.G. Castellano, P.A. Castillo, y G. Romero, Algoritmos Genéticos Distribuidos Usando SOAP, publicado en las Actas XII Jornadas de Paralelismo, Universidad Politécnica de Valencia, 2001, pp. 99-104.

[9] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs, Springer-Verlag, 1992.

[10] E. Alba, Análisis y Diseño de Algoritmos Genéticos Paralelos Distribuidos, Tesis Doctoral, Universidad de Málaga, Marzo 1999, Capítulo 1, pp. 2-14 .

[11] Christian Veenhuis , Katrin Franke , y and Mario Köppen. A semantic model for evolutionary computation. En Proceedings IIZUKA, 2000, pp. 68-73.

[12] Sitio Web de EAML. Disponible en `http://vision.fhg.de/~veenhuis/EAML/intro.html`.

[13] Juan J. Merelo-Guervós . OPEAL, una librería de algoritmos evolutivos en Perl, publicado en las Actas del Primer Congreso Español sobre Algoritmos Evolutivos y Bioinspirados, AEB'02, Mérida, Febrero 2002, pp. 54-59.

[14] Juán J. Merelo-Guervós , Pedro Ángel Castillo Valdivieso, y Javier García Castellano. Algoritmos evolutivos P2P usando SOAP . Publicado en las Actas del Primer Congreso Español sobre Algoritmos Evolutivos y Bioinspirados, AEB'02, Mérida, 2002, pp. 31-37.

[15] Vladimir Filipovic. Evolutionary Algorithm Web Service. Disponible en `http://www.matf.bg.ac.yu/~vladaf/EaWeb/index_e.html`. Junio 2002.

[16] David Gjerdingen. XmlMen Project Report. Disponible en el wiki `http://tyvex.mrs.umn.edu/twiki/view/CSci4553/XmlMenProjectReport`. Mayo 2002.

[17] Andreas Rummler. EALib - a Java Evolutionary Computation Toolkit. Disponible en la página web del autor `http://www.evolvica.org/ealib/index.html`. 2002.

[18] P. Bosman. Evolutionary Algorithm Visualization - EAVisualizer GUI. Disponible en `http://www.cs.uu.nl/people/peterb/computer/ea/eavisualizer/EAVisualizer.html`. 2001.

[19] Evolvica, Department of Electronic Circuits & Systems of the Technical University of Ilmenau. Disponible en `www.evolvica.org`. 2002.

[20] Sitio Web de Eclipse. Disponible en `http://www.eclipse.org`.

[21] E. Alba, J. G. Nieto. Optimization Algorithm DTD v.2, Data Tipe Document. Disponible en `http://tracer.lcc.uma.es/ros/documents/optimization_algorithm.dtd`, 2004.

[22] E. Alba, J. G. Nieto, A. J. Nebro. On the Configuration of Optimization Algorithms by Using XML Files, Informe Técnico. Disponible en `http://tracer.lcc.uma.es/ros/documents/xml-config.pdf`. Marzo 2003.

[23] Frank Vavak and Terence C. Fogarty.: Comparison of Steady State and Generational Genetic Algorithms for Use in Nonstationary Environments,Faculty of Computer Studies & Mathematic, University of the West of England, Bristol BS16 1QY, UK, 1996, 1-3.

[24] Alba E. and Troya J.M.: Cellular Evolutionary Agorithms: Evaluating the Influence of Radio, Proceedings of the Parallel Problem Solving from Nature VI, Schoenauer M. et al, 2000, 29-38.

[25] E. Alba and J.M. Troya.: An Analysis of Synchronous and Asynchronous Parallel Distributed Genetic Algorithms with Structured and Panmictic Islands, Parallel and Distributed Processing, 1999, 1-5.

[26] T.Back and F.Hoffmeister and H.Schewefel.: A Survey of Evolution Strategies, Dpt. of Computer Science XI, University of Dortmund, D-4600 , Dortmund 50, Germany, 1991.

[27] Theodore W. Manikas and James T. Cain.: Genetic Algorithms vs. Simulated Annealing: A Comparison of Approaches for Solving the Circuit Partitioning Problem, University of Pittsburgh, Dept. of Electrical Engineering, 1997, 1-4.

[28] P. Isasi, C. Luque, J.Hernandez, J. Valls.: Predicciones de series temporales (mareas de Venecia) mediante algoritmos genéticos, informe técnico del projecto Tracer. Disponible en `http://tracer.lcc.uma.es/problems/tide/tide.html`.

[29] J.D. Schaffer and L.J. Eshelman.: Proceedings of the 4th International Conference on Genetic Algorithms: On Crossover as an Evolutionary Viable Strategy, R.K. Belew and L.B. Booker, Morgan Kaufmann, 1991, 61-68.

[30] A. Törn and A. Zilinskas.: Global Optimization: Global Optimization, 1989, volume 350.

[31] Etherape, a Graphical Network Monitor. Disponible en `http://etherape.sourceforge.net/`.