Framework de Servicios Web para dispositivos embebidos. Orquestación de servicios

Virgilio Gilart-Iglesias, Diego Marcos-Jorquera, Ildefonso Mas-Fernández, Carlos Martínez-Martínez, Luis Felipe Herrera-Quintero, Francisco Maciá-Pérez

Universidad de Alicante. Departamento de Tecnología Informática y Computación
{vgilart, dmarcos, imas, cmartinez2, lfherrera, pmacia}@dtic.ua.es
http://www.dtic.ua.es

Resumen. En el presente artículo se propone un framework de Servicios Web para dispositivos embebidos con limitaciones de memoria y procesamiento, que permite la encapsulación de la funcionalidad del dispositivo como servicio, su composición mediante BPEL, su publicación y su posterior ejecución y consumo. Para ello se describen los diferentes módulos que se han desarrollado: módulo SOAP, módulo UDDI, motor BPEL y módulo de despliegue. Además se proponen dos escenarios donde se ha validado la propuesta, utilizando para ello el dispositivo embebido *XPort* como prototipo.

1 Introducción

En la actualidad el uso de sistemas embebidos se está convirtiendo en una de las tecnologías más extendidas en todos los ámbitos de la sociedad. Estos dispositivos ofrecen diversas ventajas como la incorporación de capacidades de cómputo y comunicación a sistemas cotidianos, lo que se conoce como computación ubicua, o el soporte a sistemas y servicios informáticos especializados con características de robustez, bajo coste, bajo consumo y facilidad de uso y gestión.

La aparición de estos sistemas embebidos abre un abanico de posibilidades que permite combinar la funcionalidad de estos elementos, junto con el resto de sistemas e infraestructuras TIC, para lograr sistemas de más alto nivel con un objetivo previamente definido, como por ejemplo sistemas domóticos, sistemas de seguridad y vigilancia, etc.

Sin embargo, en este tipo de escenarios más sofisticados, el principal problema que surge es la complejidad en la gestión e integración de todos estos dispositivos y servicios para cumplir los requerimientos del sistema general. Para su adecuado funcionamiento sería necesario el uso de modelos de gestión ágil y flexible que permitan adaptar el sistema y la funcionalidad del mismo a las diferentes configuraciones y cambios del entorno.

En función de estos requerimientos, una de las actuales propuestas que más éxito está teniendo en el ámbito de las TIC para la gestión e integración es el uso del paradigma de *Arquitectura Orientada a Servicios* (SOA) llevado a cabo mediante el uso de *Servicios Web* (Web Services – WS). En este sentido SOA y WS facilita la

composición de servicios en un entorno heterogéneo y con un alto grado de desacoplamiento.

El objetivo de la propuesta es establecer un framework para incorporar SOA y WS en dispositivos embebidos y de esta forma poder mostrar la funcionalidad de los dispositivos embebidos como WS, publicar estos servicios en un registro UDDI y componer dichos servicios mediante el uso de un motor BPEL embebido.

Para desarrollar la propuesta en el siguiente apartado se analizan las tecnologías y trabajos relacionados, a continuación se describe el framework implementado para un dispositivo embebido, posteriormente se presentan dos escenarios diferentes que permitan validar la propuesta y por último se exponen las conclusiones del trabajo.

2 Estado del Arte

Los Servicios Web son actualmente la tecnología más utilizada para el desarrollo de aplicaciones dentro de un paradigma SOA. Esto conlleva el uso de varias tecnologías como WSDL [1], SOAP, UDDI [2], y BPEL [3].

El Lenguaje de Descripción de Servicios Web (WSDL) [4] es una especificación que fue creada para describir y publicar de una forma estándar los formatos y protocolos de un servicio web. Estos estándares de interfaz son necesarios para evitar crear interacciones especiales con cada proveedor de servicios en la red. WSDL está basado en XML [5] y contiene una descripción de los datos que se tienen que pasar a un servicio web. De esta forma, tanto el emisor como el receptor conocen los datos que tienen que intercambiar para llevar a cabo la comunicación. Los elementos WSDL también contienen una descripción de las operaciones que se realizan sobre esos datos y una referencia al protocolo o al transporte empleados. Con esto el emisor sabe como enviar los datos y el receptor como procesarlos. Normalmente WSDL se utiliza con SOAP y la especificación WSDL incluye una referencia (*binding*) SOAP.

Una hoja WSDL la podemos dividir conceptualmente en dos partes: la interfaz y la implementación del servicio. En la primera de ellas se identifica los contenidos y los tipos de datos de los mensajes, así como las operaciones que se llevan a cabo con esos mensajes. Y contiene las siguientes estructuras (qué operaciones provee el servicio y como se invocan):

- *Types*. En forma de esquemas XML se definen los tipos de datos empleados para describir los mensajes intercambiados.
- *Message*. Definición abstracta de los datos que se intercambian en las operaciones. Un mensaje se divide en una serie de partes lógicas, cada una de las cuales se asocia con alguna definición de algún sistema de tipos.
- *PortType*. Conjunto abstracto de operaciones, cada una de las cuales hace referencia a un mensaje de entrada y uno de salida.
- *Binding*. Especifica un protocolo concreto y el formato de los datos de las operaciones y los mensajes definidos por un *portType* en concreto.
- Y en la implementación se incluyen las referencias a los protocolos o transportes empleados para el intercambio de mensajes (dónde se ubica el servicio).

- *Service*. Una colección de puertos relacionados. Hacen corresponder el *binding* al puerto e incluyen definiciones de extensibilidad.
- *Port*. Especifica una dirección para un *binding*, definiendo un único modo de comunicación.

SOAP, es un protocolo ligero basado en XML para el intercambio de información en un entorno distribuido [4]. Facilita la intercomunicación entre objetos de cualquier tipo, sobre cualquier plataforma y en cualquier lenguaje. SOAP puede emplear como transporte cualquier protocolo de aplicación, como por ejemplo, HTTP o SMTP. La estructura de un mensaje SOAP se puede ver en la figura 1.

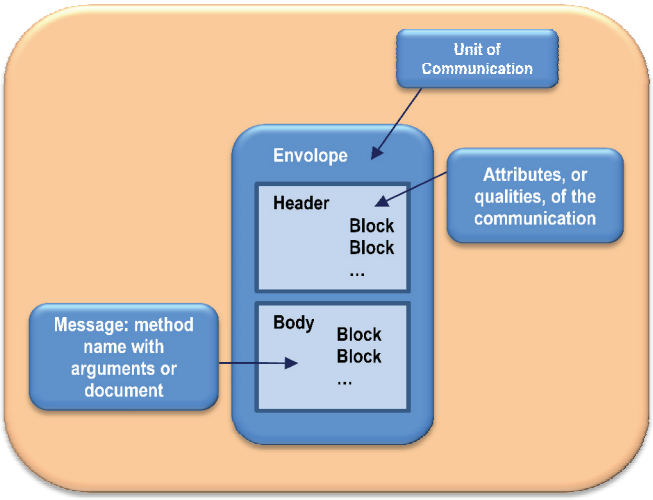

Fig. 5. Estructura del mensaje SOAP.

UDDI (Integración, Descubrimiento y Descripción Universal) [4] permite a las empresas registrarse en un tipo de directorio de páginas amarillas que les ayuda a anunciar sus servicios. De esta forma, las entidades pueden encontrase unas a otras y realizar transacciones en la Web. El proceso de registro y consultas se realiza utilizando mecanismos basados en XML y HTTP.

Un registro UDDI tiene tres secciones conceptuales:

- Blanca. Similar a la información que aparece en un directorio telefónico, que incluye nombre, teléfono y dirección.
- Amarilla. Como las páginas amarillas, e incluyen categorías de catalogación industrial, ubicación geográfica, etc.
- Verde. Información técnica acerca de los servicios ofrecidos por los negocios.

Las estructuras que se emplean son los siguientes:

- *businessEntity*. Información sobre un negocio o entidad. Utilizada por el negocio para publicar información descriptiva sobre sí mismo y los servicios que ofrece.

- *businessService*. Servicios o procesos de negocios que provee la estructura *businessEntity*.
- *bindingTemplate*. Datos importantes que describen las características técnicas de la implementación del servicio ofrecido.
- *tModel*. Especificación técnica para llevar a cabo las tareas de descubrimiento y categorización de servicios web.
- Cualquier registro UDDI debe presentar dos categorías de API:
- De publicación. Mecanismo para que los proveedores de servicios se registren (ellos mismos y sus servicios) en el registro UDDI.
- De consulta. Permite a los subscriptores buscar los servicios disponibles y obtener el servicio una vez localizado.

Otra tecnología a tener en cuenta es el Lenguaje de Ejecución de Procesos de Negocio (BPEL) lo cual se tiene el fin de componer los servicios web. En esencia este lenguaje lo utilizamos para el control centralizado de la invocación de diferentes servicios web, con cierta lógica de negocio añadida que permite la programación a gran escala. Es decir, una definición de un proceso BPEL usa uno o más servicios descritos por interfaces WSDL y proporciona una descripción del comportamiento y las interacciones de la instancia de un proceso relativa a sus componentes y recursos a través de esas interfaces. En la figura 2**Fig. 6** podemos observar el esquema conceptual de un proceso BPEL.

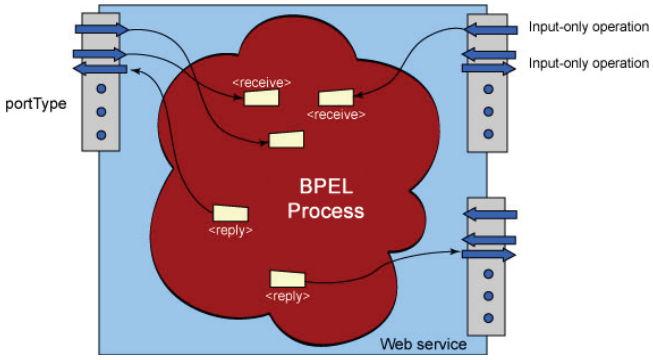

Fig. 6. Esquema de un proceso BPEL.

Las diferentes secciones de una hoja BPEL que describe un proceso de negocio son las siguientes:

- *PartnerLinks*. Define las diferentes partes que interactúan con el proceso de negocio. Cada *partnerLink* se caracteriza por un *partnerLinkType* con uno o dos roles. Esta información identifica la funcionalidad que debe ser proporcionada por el proceso de negocio y la que debe proporcionar cada servicio externo para poder establecer las relaciones con éxito. Para ello en las hojas WSDL de cada servicio que forme parte del proceso de negocio se definirán los *partnerLinkTypes* como un enlace al *portType* implicado. Y será el propio proceso de negocio BPEL el encargado de definir las referencias a estos últimos través de los *partnerLinks* (figura 3).

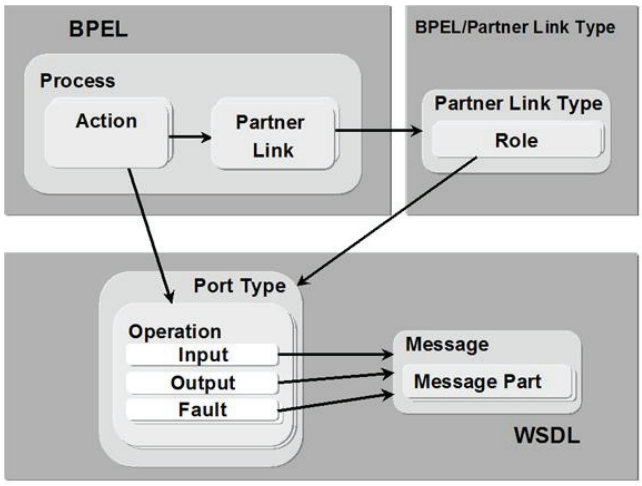

Fig. 7. Modelo BPEL y su relación con WSDL.

- *Variables*. Variables de datos utilizadas por el proceso, proporcionando sus definiciones en términos de los mensajes de las interfaces WSDL. Permiten que el proceso mantenga el estado en el intercambio de diferentes mensajes.
- *faultHandlers*. Actividades de corrección de errores.
- *process*. Descripción del comportamiento normal para manejar una petición sobre el proceso.

El proceso BPEL por si solo es básicamente un diagrama de flujo expresando un algoritmo. Cada paso en el proceso recibe el nombre de actividad, y hay una colección de actividades primitivas, como por ejemplo, invocar una operación de algún servicio web (<*invoke*>), esperar la invocación por alguien externo de una operación de la interfaz del servicio web que representa el proceso (<*receive*>), generar la respuesta a una operación de entrada/salida (<*reply*>), esperar por un tiempo (<*wait*>), copiar datos desde un lugar a otro (<*assign*>), etc.

Las actividades primitivas pueden combinarse en algoritmos más complejos empleando alguna de las estructuras proporcionadas por el lenguaje. Podemos definir una secuencia ordenada de pasos (<*sequence*>), tener secuencias condicionales como <*switch*>, definir bucles (<*while*>), ejecutar varios pasos en paralelo (<*flow*>), etc. De esta forma se pueden combinar recursivamente las actividades estructuradas para expresar algoritmos de diferente complejidad que representen las implementaciones del servicio.

Otro aspecto a tener en cuenta es cómo dar de alta un servicio representado por una interfaz WSDL y un proceso descrito en una hoja BPEL en un registro UDDI. Para ello debemos encontrar una correspondencia adecuada entre los lenguajes WSDL/BPEL y UDDI. En las figuras 4 y 5 se puede observar el esquema de la correspondencia entre una hoja BPEL y WSDL [6][7] en una estructura de elementos UDDI.

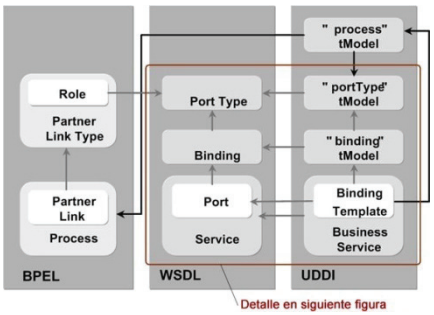

Fig. 8. Correspondencia BPEL en UDDI.

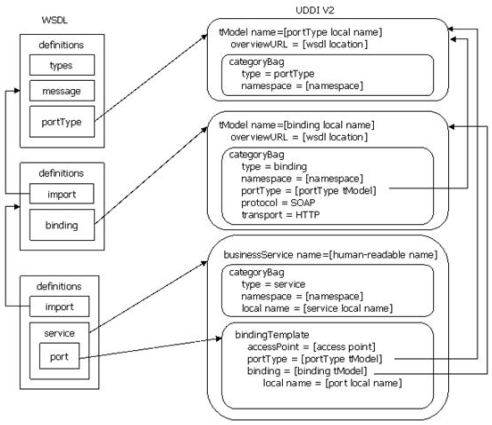

Fig. 9. Correspondencia WSDL en UDDI.

Como se observa la correspondencia utiliza varias entidades UDDI para representar las entidades en los documentos WSDL y BPEL. Por lo que requerimos un mecanismo para indicar qué tipo de entidad WSDL o BPEL está siendo descrita por cada elemento UDDI. Para ello se emplea la entidad UDDI *tModel*, junto con una serie de catalogadores. En la Tabla 1 podemos ver las categorías empleadas para este propósito.

Tabla 5. Categorías empleadas en la correspondencia BPEL-UDDI

Nombre	Descripción	Key
uddi.org:bpel:types	Tipos BPEL	uuid:e8d75f6c-3f24-3b8d-97fd-f168e424056f
uddi-org:wsdl:types	Tipos WSDL	uuid:6e090afa-33e5-36eb-81b7-1ca18373f457
uddi-org:xml:namespace	Espacios de nombres	uuid:d01987d1-ab2e-3013-9be2-2a66eb99d824
uddi-org:xml:localName	Nombres locales XML	uuid:2ec65201-9109-3919-9bec-c9dbefcaccf6
uddi-org:wsdl: portTypeReference	Referencias a tModels que representan entidades wsdl:portTypes	uuid:082b0851-25d8-303c-b332-f24a6d53e38e
uddi-org:protocol:soap	tModels que representan el protocolo SOAP 1.1	uuid:aa254698-93de-3870-8df3-a5c075d64a0e
uddi-org:protocol:http	tModels que representan el protocolo HTTP	uuid:6e10b91b-babc-3442-b8fc-5ª3c8fde0794

3 Framework de Servicios Web para dispositivos embebidos

En el presente apartado se describe los diferentes módulos que componen el framework de Servicios Web para dispositivos embebidos (ver figura 6). Previamente se describe el escenario general SOA donde la propuesta será llevada a cabo. Además se describe la plataforma física sobre la que se ha desarrollado la propuesta del presente trabajo.

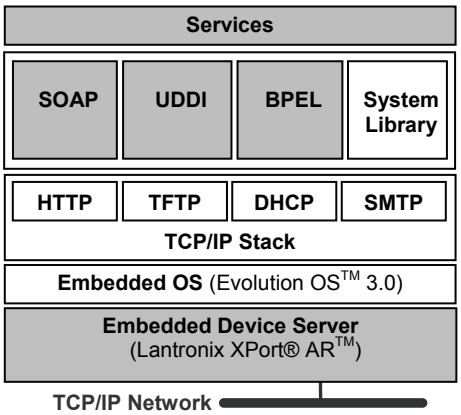

Fig. 6. Arquitectura del prototipo

3.1 Esquema General

Para el presente trabajo se ha propuesto un escenario general basado en la arquitectura SOA en el que los dispositivos embebidos se comportan como proveedores de servicios (ver figura 7). Los servicios que se ofrecen podrán ser compuestos para lograr una funcionalidad de mayor nivel que será consumida por clientes WS estándar. Tanto los servicios atómicos como los compuestos deberán ser publicados por el dispositivo en un registro UDDI para permitir su localización por parte del cliente y, de esta manera, poder ser consumido desde cualquier *WS Client*. La composición de los servicios el dispositivo podrá ser realizadas a través del *Management System* y su despliegue será realizado mediante un servicio específico actuando en este caso el *Management System* como un consumidor de servicios.

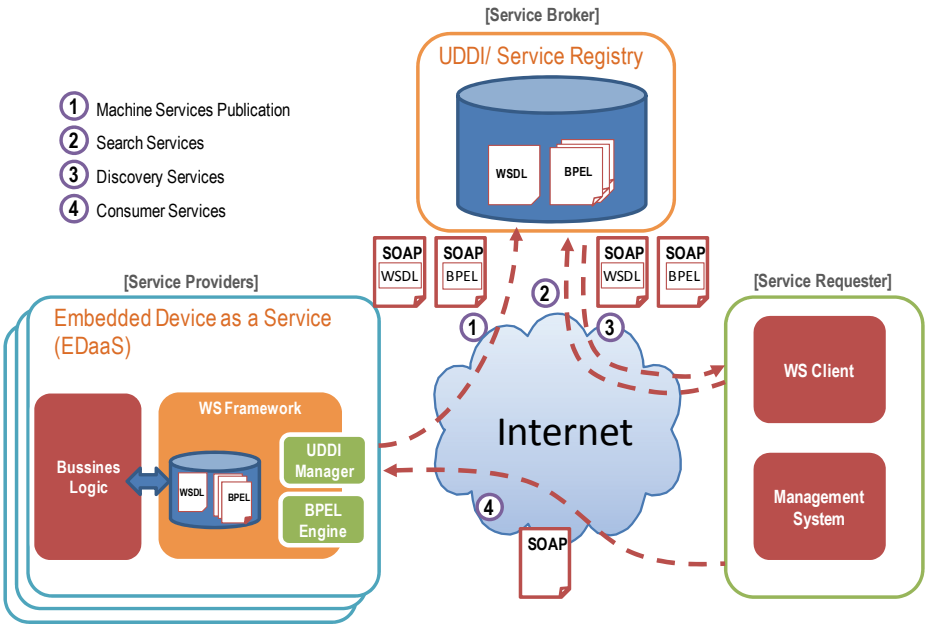

Fig. 7. Esquema de funcionamiento del sistema propuesto.

3.2 Dispositivo Físico

Para la implementación del prototipo se ha seleccionado un dispositivo embebido genérico de la marca Lantronix denominado *XPort* [8] que posee unas características muy adecuadas para la elaboración de la plataforma. El *XPort* es un servidor web embebido que tiene las características de un microordenador que permite la redirección de datos desde Ethernet al puerto serie [9]. Sus características técnicas se presentan en la Tabla 2.

Tabla 2. Características técnicas del Xport.

CPU, Memoria	Microprocesador de 16 bits Lantronix DSTni-EX 186 CPU, 256 KB zero wait state SRAM 512 KB Flash, 16 KB Boot ROM
Firmware	Actualizable por TFTP y puerto serie
Interfaz Serial	CMOS (Asíncrono) Niveles de Voltaje de 3.3V Tasa de Transferencia (300 bps to 921600 bps)
Formato Serial	7 ó 8 bits de datos, 1-2 bits de parada, Paridad: Par, Impar, ninguna
Control de Flujo	XON/XOFF (software), CTS/RTS (hardware), ninguna
Pines de I/O	3 PIO pines (Seleccionables por software) 4mA max.
Interfaz de Red	RJ45 Ethernet 10BASE-T o 100BASE-TX (auto-detección) Vers 2.0/IEEE 802.3
Protocolos Suportados	ARP, UDP/IP, TCP/IP, Telnet, ICMP, SNMP, DHCP, BOOTP, TFTP, Auto IP, and HTTP
Administración	Servidor web interno, SNMP, Serial, Telnet
Security	Protección por Contraseña, Encriptación Rijndael 128-, 192-, ó 256-bit
Servidor Web	Capacidad de almacenamiento de páginas web 384 KB
Peso	9.6 gramos

XPORT

Este dispositivo se ha seleccionado por varias razones, como son: bajo coste, alta velocidad de respuesta, pasarela Ethernet serie, manejo de lenguaje C estándar para su programación, capacidad de manejar diversos protocolos de red como TCP, UDP, DHCP, etc.

3.3 Módulo SOAP

Ésta es la capa central del framework, en ella se apoyan los servicios para su funcionamiento, mientras que ésta a su vez se ayuda del módulo UDDI para la publicación de los mismos. La capa SOAP se encargará de abstraer la implementación SOA tomada traduciendo los mensajes de entrada y salida de los servicios utilizados por la implementación propuesta. En este caso la traducción se realizará a paquetes SOAP ya que utilizaremos Servicios Web.

3.3.1 Tablas

Para llevar a cabo la gestión de los servicios es necesario almacenar los métodos que ofrecen éstos. Para hacer esto se han utilizado las estructuras que aparecen en la figura 8.

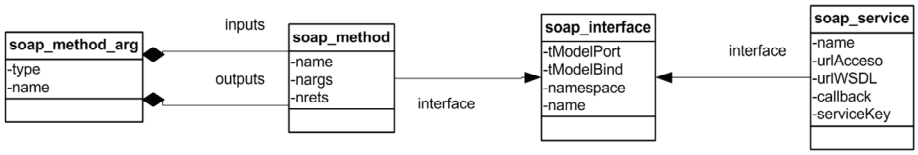

Fig. 8. Estructuras SOAP para almacenar los servicios y sus métodos.

Como se puede apreciar en la figura, se utiliza el concepto de interfaz, a modo de simplificar en la medida de lo posible el almacenamiento de los servicios. Esto quiere decir que se puede introducir una interfaz con sus métodos y más tarde introducir varios servicios que implementen la interfaz y por lo tanto todos los métodos de esta. En cuanto a los métodos resaltar el hecho de que pueden recibir y retornar múltiples parámetros de tipos simples, es decir, booleano, enteros y cadenas de texto.

En los siguientes puntos se tratará con más detenimiento como se definen las interfaces y como el servicio debe implementar los métodos de esta.

En lo referente a las tablas que utiliza el módulo SOAP para su correcto funcionamiento queda mencionar una estructura llamada *soap_method_data* que consiste en dos listas de atributos, que serán utilizados para enviar los inputs y recibir los outputs durante la llamada a un servicio.

3.3.2 Interfaz

El módulo SOAP ofrece los siguientes métodos para crear un servicio específico:

soap_interface_register → Con este método el módulo SOAP se comunicará con el módulo UDDI para registrar la interfaz en el directorio. Una vez la interfaz se ha registrado en el directorio UDDI se almacena en las tabla *soap_interface* junto con su espacio de nombres y su nombre. De momento permanecerá registrada sin métodos.

soap_method_register → Esto añade un método a una interfaz. Esto no tiene repercusiones directas en UDDI. Este método inicialmente no tendrá parámetros ni de entrada ni de salida.

soap_method_add_arg → Dado un método de una interfaz se añade un parámetro de entrada. Los parámetros pueden ser de tres tipos: entero, cadena y booleano.

soap_method_add_ret → Dado un método de una interfaz añadimos un parámetro de salida. Los parámetros pueden ser de tres tipos: entero, cadena y booleano.

soap_service_register → Crea un servicio dentro del sistema. Este método realiza la comunicación pertinente con la capa UDDI para registrar el servicio en el UDDI. Para dar de alta el servicio es necesario conocer el nombre del servicio, la URL con la que se quiere acceder al servicio, la URL con la que se quiere acceder a la página WSDL, la interfaz que implementa el servicio y un acceso a la implementación. Con esto simulamos en nuestro sistema en lenguaje C la implementación de interfaces de otros lenguajes más modernos como Java o C#. Es decir, para dar de alta un servicio por un lado se tiene que registrar la interfaz y por otro hay que definir la implementación de la misma. Esta implementación será un puntero a función, que nos proporcionará la entrada y salida al servicio, utilizando la estructura *soap_method_data* antes explicada.

Por otro lado, esta capa ofrece otros métodos para simplificar otras tareas:

soap_call → Este método será lanzado cuando un servicio sea invocado por un cliente. Se encargará de leer la petición SOAP del paquete de entrada, rellenar a partir de esta la estructura *soap_method_data* y realiza la llamada al servicio adecuado. Una vez el servicio responde, el método compone el mensaje de respuesta y este es enviado de vuelta al cliente.

wsdl_callback → Este método será lanzado cuando un cliente solicite el descriptor WSDL de un servicio. El método generará dinámicamente dicho descriptor y se lo retornará al cliente.

3.3.3 Funcionamiento

Para ver el funcionamiento de este módulo a continuación se muestran algunos ejemplos. El código de las siguientes imágenes pertenece al que se debería escribir en un servicio que se quisiera implementar. En la figura 9 se puede ver un ejemplo de cómo definir una interfaz sencilla invocando los métodos de SOAP:

```
/*Registro la interface*/
char s1[]="es.ua.dtic.bases";//Espacio de nombres
char interfaz[]="IFactory";//Nombre de la interfaz
interface=soap_interface_register(s1,interfaz);//Creamos la interfaz
if (interface == -1) return -1;
//Método obtener sensores
char f3[]="Sensores";//Nombre del metodo
if ((mth=soap_method_register(interface, f3)) == -1) return -1;//Introducimos el metodo
for(i=0;i<base_num_sensores;i++){//Introducimos cada como sensor como un parametro de retorno
    if (soap_method_add_ret(mth, SensoresControl[i].name, SOAP_DATA_TYPE_BOOLEAN) == -1) return -1;
}
...Más métodos
```

Fig. 9. Definición de la interfaz

Después se tendrá que definir la función que implemente la interfaz como se puede ver en la figura 10.

```
int imaas_base3_callback(char *name, struct soap_method *method, struct soap_method_data *data)
{
    int arg;

    //Comprobamos a que método del servicio se quiere invocar y realizamos la llamada a la implementación

    if (strcmp(method->name, f1) == 0)//Arrancar motor
    {
        MoverMotor(data);
        return 0;
    }
    if (strcmp(method->name, f2) == 0)//Detener motor
    {
        DetenerMotor(data);
        return 0;
    }
    if (strcmp(method->name, f3) == 0)//Estados sensores
    {
        ObtenerSensores(data);
        return 0;
    }
    if (strcmp(method->name, f4) == 0)//Estado máquinas
    {
        ObtenerMotores(data);
        return 0;
    }

    ...Más comprobaciones
```

Fig. 10. Implementación del servicio

Ahora es necesario hacer la llamada para registrar el servicio asociado a esta implementación. En la figura 11 se puede ver esta llamada.

```
                //nombre servicio, ipAcceso, dirWSDL, interfaz, función implementación
srv = soap_service_register(serv,ipe,serv,interface, imaas_base3_callback);
if (srv == -1) return -1;
```

Fig. 11. Registro del servicio

Una vez se ha registrado el servicio ya se puede consultar su WSDL generada dinámicamente y realizar peticiones al servicio web. Se puede ver la página WSDL de un servicio en la figura 12.

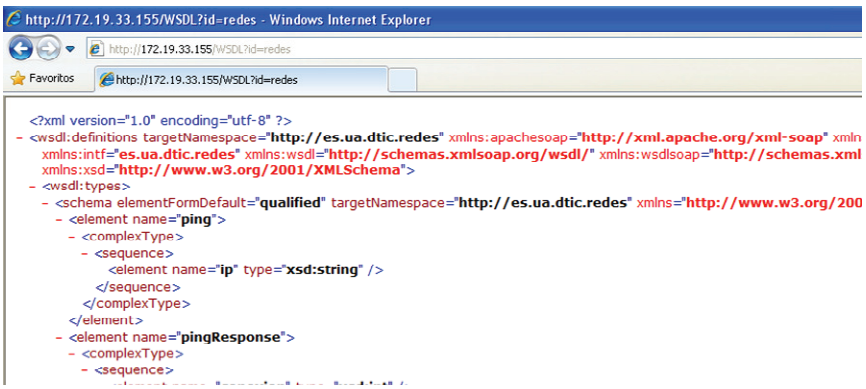

Fig. 12. Visualización del descriptor WSDL de un servicio generado dinámicamente

3.3.4 Interfaces gráficas

El módulo SOAP también implementa una interfaz gráfica para su monitorización. Dado que SOAP se encarga simplemente de gestionar las interfaces y los servicios, ofrece dos direcciones web de consulta. El primero es *interfaces.htm* que se muestra en la figura 13 y el segundo es el *services.htm* que se muestra en la figura 14.

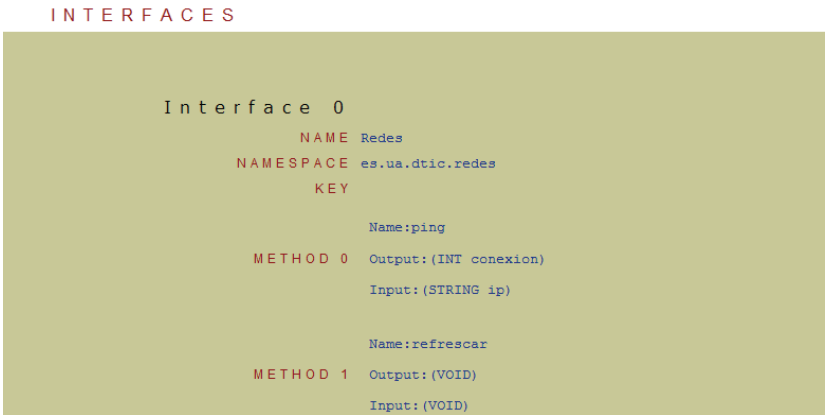

Fig. 13. Interfaz gráfica que muestra un listado de interfaces

En la interfaz *interfaces.htm* se puede ver un listado de las interfaces que se encuentran definidas en el sistema. Para cada interfaz se puede ver los métodos definidos, y por cada uno de estos su nombre, entrada(s) y salida(s).

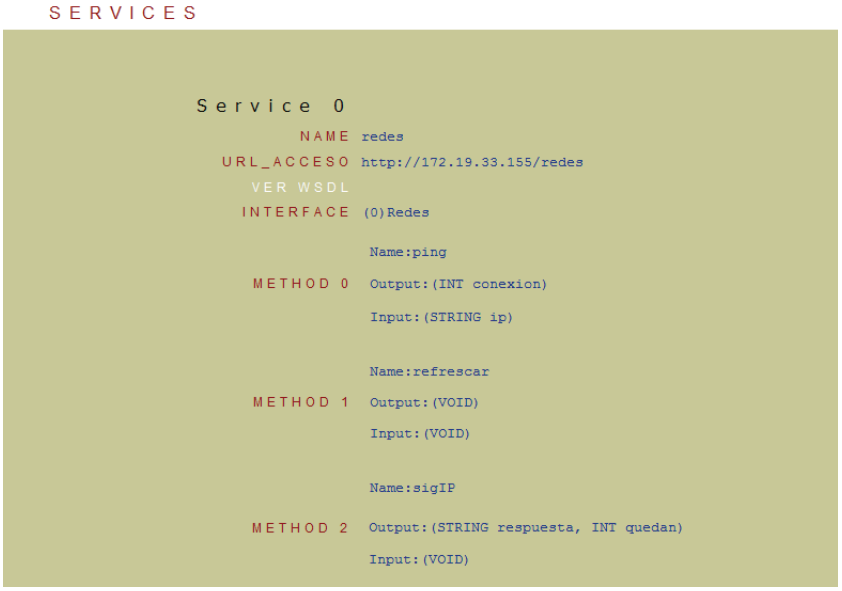

Fig. 14. Interfaz gráfica que muestra un listado de servicios

En la interfaz *services.htm* se puede ver un listado de los servicios que se encuentran publicados en el sistema. Para cada servicio se puede ver sus características principales, la interfaz que implementan y los métodos de manera similar a como se mostraban en la interfaz anterior.

3.4 Modulo de Publicación

El componente publicación tiene varios requisitos previos. Se debe conocer qué negocio tratamos, tanto su nombre como el *businessKey* del negocio al que pertenecen sus servicios, y la dirección IP del servidor UDDI (donde debe existir dicho *businessKey*). Utilizando este *businessKey* y consumiendo los servicios web ofrecidos por el registro UDDI, el módulo de publicación puede registrar los servicios. Ésto lo hace en cuatro pasos:

1. Obtener el *token* del registro UDDI.
   No se puede operar con el UDDI sin una previa autenticación del usuario para realizar operaciones. Para ello se solicita un *authToken* al UDDI utilizando el nombre del negocio, y dirigiéndonos a la IP en la que sabemos que reside el UDDI. Para ello se dispone de una función que devuelve esa clave para las sucesivas operaciones.

2. Registrar la interfaz
   a. Registrar en UDDI el *portType* de la interfaz WSDL como un *tModelPortType*.
   Para este paso necesitamos el *authToken*, el nombre de la interfaz, la URL de acceso y su espacio de nombres. Se obtiene el *tModelKey* del *portType*.

   b. Registrar en UDDI el *binding* de la interfaz.
   Se complementará el registro haciendo el *tModelBinding*, el cual referencia al *portType* y servirá para ser enlazado desde los servicios. Se obtiene el *tModelKey* del *Binding*

3. Registrar el servicio en el UDDI.
   Por cada servicio que se tenga, se publicará su nombre y *namespace* correspondiente. Obtenemos el *serviceKey*.

4. Registrar el *binding*. Asociación de servicio, *portType* y *binding*.
   Finalmente, se asocia el servicio, interfaz (*tModelPortType* y *tModelBinding*), URL de acceso al servicio y el nombre local del servicio, obteniendo un *bindingKey*.

3.4.1 Funcionamiento

Debido a las limitaciones de memoria del dispositivo *XPort*, no se pueden almacenar todos los mensajes SOAP necesarios para realizar toda la publicación en el UDDI. De este modo, se ha optado por situar como fichero externo (en la memoria flash del dispositivo) cada uno de los mensajes SOAP necesarios, con etiquetas del estilo <#...#> en los lugares donde deben ser sustituidos por los valores adecuados según el servicio. A continuación se muestra un ejemplo sencillo, como un mensaje de petición de un *authToken*, donde se necesita únicamente el identificador del *publisher*.

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
  <soapenv:Body>
    <get_authToken generic="2.0" xmlns="urn:uddi-org:api_v2"
      userID="<#Publisher-id#>"/>
  </soapenv:Body>
</soapenv:Envelope>
```

Otros mensajes son mucho más complejos y requieren de más información, pero por cuestiones de espacio no se reflejan en este trabajo.

3.4.2 Interfaz de uso del módulo UDDI

Se ha procurado encapsular las funcionalidades para representar el funcionamiento que desde la consola de UDDI se sigue a la hora de publicar servicios. De este modo, a la hora de publicar, se pueden seguir sencillamente los pasos de publicación mediante las llamadas a las funciones correspondientes:

```
int get_authToken(…);
int save_tModel(…);
int save_service(…);
int save_binding(…);
```

3.5 Módulo de motor BPEL

Uno de los principales problemas que se encuentran a la hora de programar un servicio en el *XPort* es la limitación de su memoria. En este caso se pone de manifiesto debido al gran tamaño de las hojas BPEL y es por ello que se restringe la sintaxis de las páginas BPEL. En los sucesivos apartados explicaremos qué sintaxis deben tener los documentos WSDL y BPEL para poder utilizarlos sobre el motor BPEL.

3.5.1 Módulo de despliegue

En la propuesta se han desarrollado dos formas de desplegar una hoja BPEL para su ejecución en el dispositivo. Como se pretendía seguir la filosofía SOA, se ha implementado un servicio web para la recepción de la hoja a ejecutar. Por las limitaciones del dispositivo este método puede conllevar problemas en el caso de páginas BPEL de gran tamaño. Es por ello que alternativamente también se ha implementado un despliegue en el que se pasa la URL donde se encuentra alojada la hoja BPEL. Aprovechando las funcionalidades del dispositivo *XPort*, se ha creado un proceso paralelo que periódicamente comprueba si se hay alguna URL pendiente de procesar. Una vez ha detectada una URL, se solicita y transfiere la hoja BPEL para su posterior procesamiento.

Una vez desplegada la hoja BPEL también es publicada en el registro UDDI, y será ofrecida como servicio.

3.5.2 Formato de WS para añadir hojas BPEL

Definidas las dos maneras de desplegar hojas BPEL (*AddHojaBPEL* y *AddHojaBPEL2*), en las figuras 15 se puede observar cómo sería el contenido de los paquetes SOAP para invocar al servicio de despliegue.

```
<AddHojaBPEL>
    <proceso>
        Código BPEL
    </proceso>
</AddHojaBPEL>
```

```
<AddHojaBPEL2>
    <urlBPEL>
        URL a hoja BPEL
    </urlBPEL>
</AddHojaBPEL2>
```

Fig. 15. Invocación a servicios de despliegue de páginas BPEL.

Una vez invocado el servicio de despliegue, se procederá a su carga. Para ello se dispone de una serie de autómatas muy ligeros que procesarán la hoja conforme a su lectura. A razón de este procesamiento la sintaxis de los documentos debe ser tal y como se especifica, siguiendo el orden e introduciendo todos los atributos (aunque éstos sean después desechados). La sintaxis de las hojas no será comprobada y el Xport probablemente no funcione si ésta no es correcta. En la definición de la sintaxis utilizaremos los siguientes caracteres:

- '*' Cadena que se desecha
- '---' Cadena que se leerá y tratará
- '[]' Notas y aclaraciones

Por último, también hay que hacer notar que las hojas WSDL creadas dinámicamente por Eclipse y Visual Studio, funcionan correctamente, ya que la propuesta se ha basado en dicha especificación para crear los autómatas. Las páginas creadas con otras plataformas como *NetBeans* podrían no ser válidas. En la explicación de las hojas se ha dado por sentado la línea que la declara como un documento del tipo XML.

3.5.3 Sintaxis de las hojas WSDL

A continuación, en la figura 16, se muestra la sintaxis las hojas WSDL soportadas por la implementación.

```
<wsdl:definitions *>
    <wsdl:types>
    <schema *>
        <element name="---">
            <complexType>
    [caso de void ha de estar la etiqueta de apertura y cierre
del complexType por separado, no sirve complexType/]
            </complexType>
        </element>
    <element name="---">
    <complexType>
        <sequence>
            <element name="---" type="*:---"/>
        </sequence>
    </complexType>
    </element>
    </schema>
    </wsdl:types>
    [Aquí empiezan los message, sólo nos interesa el location así que no
los tendremos en cuenta]
    [A tener en cuenta que el nombre de los element del schema han de
corresponderse con los respectivos mensajes que los usan y éstos a su
vez con las operaciones basadas en esos  mensajes. Lo mismo ocurre para
los element-message de salida.]
*
    [el location lo hallamos en la etiqueta service]
*
location="---"/>
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

Fig. 16. Sintaxis de las hojas WSDL.

3.5.4 Sintaxis de las hojas BPEL

A continuación, en la figura 17, se muestra la sintaxis las hojas BPEL soportadas por la implementación.

```
<Process name="-----" targetNamespace="------" * >
    <import namespace="---" location="----" importType="----" */>
    <import namespace="---" location="----" importType="----" */>
    [Esperamos al menos un  import]
  <partnerLinks>
    <partnerLink name="----" location="----" partnerLinkType="---" (myRole|partnerRole|*)="---"/>
    <partnerLink name="----" location="----" partnerLinkType="---" (myRole|partnerRole|*)="---"/>
    [Esperamos al menos un partnerLink]
  </partnerLinks>
  <variables>
    <variable name="---" xmlns:tns="---" messageType="---"/>
    <variable name="---" xmlns:tns="---" messageType="---"/>
    [Esperamos al menos una variable]
  </variables>
  <sequence>
    <receive  *="*"[Se desecha el name] *="*"[Se desecha el createInstance] partnerLink="----"
operation="----" *="*"[Se desecha el xmlns:tns] *="*"[Se desecha el portType] variable="----"/>
    [Receive: Con esta actividad recibimos una petición de SOAP desde el exterior.]
    [Siempre tendremos esta secuencia que comenzará con un receive y acabará con un reply. Como
nuestro motor BPEL solamente soporta llamadas síncronas, no podrán aparecer más receives y
replies En medio de éstas podremos utilizar las funciones que explicamos en el siguiente punto]
    <reply  *="*"[Se desecha el name] partnerLink="----" operation="----" *="*"[Se desecha el
xmlns:tns] *="*"[Se desecha el portType] variable="----"/>
    [Reply: Con esta actividad respondemos al cliente que ha realizado la llamada SOAP.]
  </sequence>
</process>
```

Fig. 17. Sintaxis de las hojas WSDL.

3.5.5 Actividades BPEL contempladas

Debido a las limitaciones del dispositivo embebido no ha sido posible desarrollar todas las actividades que ofrece el lenguaje BPEL. En esta sección se va a explicar que actividades de las hojas BPEL se ha desarrollado, para que sirven cada una y la sintaxis concreta que tienen que tener para que el XPort funcione de manera adecuada.

- Actividad **invoke**: utilizada para realizar llamadas a servicios web. En la figura 18 se puede ver que sintaxis debe seguir. Lo normal es que esta actividad reciba una variable de entrada y una de salida, pero en este caso se puede omitir cualquiera de ellas.

```
<invoke *="*" partnerLink="---" operation="---" *="*" portType="----"
    inputVariable="----" outputVariable="---"/>
```

Fig. 18. Sintaxis de la actividad invoke.

- Actividad **repeatUntil**: esta actividad nos es útil para la creación de bucles dentro de los procesos BPEL. El desarrollo de esta actividad ha requerido la implementación de un lenguaje de expresiones cuya gramática se puede apreciar dentro de la sintaxis de la actividad en la figura 19.

```
<repeatUntil  *="*"[Se desecha el name]>
    <sequence>------</sequence>
    [Siempre esperaremos este sequence, que contendrá otras actividades anidadas]
    <condition>
        [Esta función contendrá una expresión regular que sigue la siguiente gramática:
        A --> not(E | E)
        E --> V | L
        L --> Var Op Var
        Op --> > | < | =
        Var --> V | int | string
        V --> $<nombre variable>.<nombre campo>
        [int y string se refieren a enteros y cadenas respectivamente]
    </condition>
</repeatUntil>
```

Fig. 19. Sintaxis de la actividad repeatUntil.

- Actividades **if**, **elseif** y **else**: estas actividades permiten tomar decisiones durante la ejecución del proceso. Estas también utilizan el lenguaje de expresiones definido para la actividad *repeatUntil*. En la figura 20 se puede ver la sintaxis de estas actividades y como deben situarse entre ellas.

```
<if *="*"[Se desecha el name]>
    <condition>---</condition><sequence>---</sequence>
    [Después del sequence pueden aparecer 0 o más elseif con el cuerpo que abajo aparece]
    <elseif>
        <condition>---</condition><sequence>---</sequence>
    </elseif>
    [Por último, después de los else-if(en caso de haberlos) puede existir un else]
    <else>
        <sequence>---</sequence>
    </else>
</if>
```

Fig. 20. Sintaxis de las actividades if, elseif y else.

- Actividad **assign**: utilizada para realizar asignación de valores a estructuras y campos de estructuras. La implementación de esta actividad ha requerido el desarrollo de un sub-lenguaje de *Xpath*. En la figura 21 se puede ver el aspecto general de esta actividad, aunque después se explicará con más detalle el contenido de las etiquetas *from* y *to*.

```
<assign *>
    <copy>
        <from >---</from>
        <to>---</to>
    </copy>
</assign>
```

Fig. 21. Sintaxis de la actividad assign.

- Actividad **from**: esta actividad irá siempre empotrada en una actividad *assign*, y dentro de ésta se encargará de marcar el origen de la copia de datos. Como BPEL soporta distintos tipos de copia la actividad *from* puede tener distintos valores. En la

figura 22 se tiene como origen un literal de cadena. Esto sirve para rellenar un campo de una variable (El *CDATA* es obligatorio)

```
<from>
    <literal><![CDATA[100]]></literal>
</from>
```

Fig. 22. Ejemplo de actividad from utilizando un literal string como origen.

Otra manera que proporciona BPEL para especificar el origen es introducir el cuerpo completo de una variable. Esto sirve para dar valor a todos los atributos de una variable a la vez. En la figura 23 se puede ver un ejemplo de este tipo de *from* (El *CDATA* es obligatorio).

```
<from>
    <persona>
        <nombre><![CDATA[nombre]]></nombre>
        <peso><![CDATA[100]]></peso>
    </persona>
</from>
```

Fig. 23. Ejemplo de actividad from utilizando un literal de variable completa como origen.

Al igual que se puede introducir el valor manualmente en la hoja para dar valor a una variable, se puede introducir una referencia a otra variable. Utilizando esta variante de la actividad se copiará una variable a otra, campo a campo. En la figura 24 se puede ver como debe ser la sintaxis de la actividad *from* cuando se quiere utilizar una variable como origen.

```
<from variable="---" *></from>
```

Fig. 24. Sintaxis de la actividad from para referenciar una variable completa..

Otra alternativa que ofrece BPEL es copiar un solo atributo de una variable, en lugar de copiarla totalmente. Para hacer esto se debe utilizar la sintaxis que aparece en la figura 25. Nótese que se necesita una expresión *XPath* para extraer el atributo de la variable.

```
<from variable="---" *>
    <query *>
        <![CDATA[expresión xpath]]>
    </query>
</from>
```

Fig. 25. Sintaxis de la actividad from cuando referenciamos un atributo como origen.

- Actividad **to**: esta actividad también se encontrará siempre empotrada en una actividad *assign* y dentro de ésta se encargará de establecer el destino de la copia de

datos. Como ya se ha visto en la actividad *from*, se ha contemplado copiar a atributos de variables o a variables completas. De esta manera la actividad *to* sólo tiene dos variantes que a continuación explicamos.

La primera opción que se tiene es utilizar como destino una variable completa. La sintaxis de esta variante la podemos ver en la figura 26.

```
<to variable="---" *></to>
```

Figura 26. Sintaxis de la actividad to utilizando una variable completa como destino.

La otra opción es especificar como destino un atributo de una variable. En la figura 27 se puede ver la sintaxis que debe seguir esta alternativa, aunque como se puede apreciar es bastante similar al de la figura 25, valiéndose del sub-lenguaje *XPath* utilizado.

```
<to variable="---" *>
    <query *>
        <![CDATA[expresión xpath]]>
    </query>
</to>
```

Fig. 27. Sintaxis de la actividad to utilizando un atributo de una variable como destino.

Una vez explicadas todas las actividades contempladas en el motor BPEL es posible crear hojas BPEL para cada servicio. En la figura 28 se puede ver un ejemplo de un hoja BPEL que el motor reconoce y ejecuta.

```
<process name="LlevarHastaFinal"
    targetNamespace="http://enterprise.netbeans.org/bpel/cinta"
    xmlns="http://docs.oasis-open.org/wsbpel/2.0/process/executable"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://enterprise.netbeans.org/bpel/cinta">
    <import namespace="http://172.19.33.241:8080"
            location="WS/perico.wsdl"
            importType="http://schemas.xmlsoap.org/wsdl/"/>
    <partnerLinks>
        <partnerLink name="Prueba1"
                     xmlns:tns="http://172.19.33.241:8080"
                     partnerLinkType="Prueba1"
                     myRole="Prueba1"/>
    </partnerLinks>
    <variables>
        <variable name="getsensor"
                  xmlns:tns="http://172.19.33.241:8080"
                  messageType="getsensor"/>
        <variable name="getsensor2"
                  xmlns:tns="http://172.19.33.241:8080"
                  messageType="getsensor"/>
        <variable name="getsensorResponse2"
                  xmlns:tns="http://172.19.33.241:8080"
                  messageType="getsensorResponse"/>
    </variables>
    <sequence>
        <receive name="Inicio" createInstance="yes"
                 partnerLink="Prueba1"
                 operation="LlevarHastaFinal"
                 xmlns:tns="http://172.19.33.241:8080"
                 portType="Prueba1"
                 variable="getsensor"/>
        <assign name="Assign1" validate="no">
                 <copy>
                     <from variable="getsensor" part="payload">
                        <query
        queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
                                <![CDATA[/tns:sensor]]></query>
                     </from>
                     <to variable="getsensor2" part="payload">
                        <query
        queryLanguage="urn:oasis:names:tc:wsbpel:2.0:sublang:xpath1.0">
                                <![CDATA[/tns:sensor]]></query>
                     </to>
                 </copy>
                 </assign>
        <reply name="Fin"
               partnerLink="Prueba1"
               operation="LlevarHastaFinal"
               xmlns:tns="http://172.19.33.246:8080"
               portType="Prueba1"
               variable="getsensorResponse2"/>
    </sequence>
</process>
```

Fig. 28. BPEL de ejemplo.

3.5.6 Proceso de lectura

Como en un principio solamente se dispone de la URL donde está la hoja BPEL, se debe solicitar al host en el que se encuentre alojada, leyéndola por completo mediante la una conexión TCP. Ahora bien, una hoja BPEL tiene una o varias WSDL que definen las interfaces, por lo que también es necesaria su lectura a la hora de poder trabajar con la hoja BPEL. Por tanto, el autómata que lee la hoja, en este momento almacena las hojas WSDL para su posterior lectura. Cuando puede pedir las WSDL, acaba de recopilar los datos necesarios (como por ejemplo la URL de acceso a los servicios) y el *XPort* se encuentra listo para ejecutar la hoja BPEL.

Tanto para posibilitar este procesado secuencial de los fuentes (hoja BPEL y hoja(s) WSDL) como para el funcionamiento del motor, se dispone de una estructura de tablas que almacenan toda la información necesaria (figura 29).

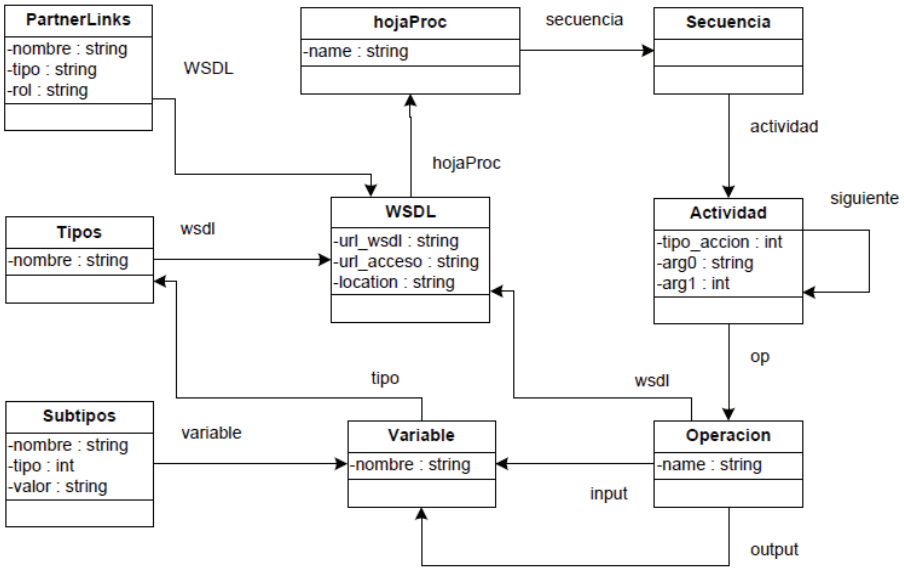

Fig. 29. Tablas de almacenamiento Motor BPEL.

3.6 Capa de servicios

En esta capa se incluyen todos los servicios que se quieren implementar. Esto quiere decir que para cada servicio se generarán nuevo sub-módulo y se utilizarán los servicios de la capa SOAP para publicarlo. En la capa SOAP se ha visto como los servicios debían utilizar los métodos que ofrecía dicha capa y en secciones posteriores se verán ejemplos de implementación de servicios.

4 Validación

Para llevar a cabo la validación de la propuesta se han definido dos posibles escenarios de aplicación, el primero orientado a dotar de capacidad de computación y comunicación a elementos de producción industrial para mostrarlos como servicios y el segundo orientado a ofrecer servicios específicos de gestión de red, ambos mediante el uso de dispositivos embebidos.

4.1 Sistema de fabricación

Esta propuesta se basa en la implementación del concepto de IMaaS [13] que permite mostrar los procesos y operaciones de fabricación llevadas a cabo por la maquinaria industrial como procesos de negocio expuestos como servicios. Este enfoque elimina las restricciones físicas y conceptuales existentes entre el nivel de empresa y los niveles inferiores de producción y, de esta forma, facilita la composición de nuevas funcionalidades expuesta como servicios. Para lograrlo, se ha dotado a la maquinaria industrial de capacidad de comunicación y computación conectando la maquinaria industrial al dispositivo embebido a través de un puerto serie.

4.1.1 Escenario de validación

Para llevar a cabo el prototipo de la plataforma mencionada se ha utilizado un escenario de simulación industrial de la empresa STAUDINGER GMHB [10] (ver figura 30), en el cual se despliegan una serie de controladores de la marca ICPDAS [11], para el manejo de los actuadores en cada subescenario, ya sea una cinta transportadora, un almacén industrial, etc.

Fig. 30. Escenario de Simulación para la maquinaria industrial.

Fig. 31. Escenario del WS Gateway para la maquinaria industrial.

Para el escenario de simulación industrial se ha implementado la interfaz que utiliza el dispositivo embebido XPORT (ver figura 31) para intercambiar mensajes con los elementos que controlan los distintos actuadores y sensores. Debido a que los controladores del escenario operan con el estándar RS485 se ha dotado al XPORT del manejo de este estándar. Las características principales del RS485 son su posibilidad de transmisión de datos a grandes distancias así como la utilización de modelos en control automático del tipo maestro/esclavo. El modelo de XPORT utilizado carece de este estándar industrial pero posee el estándar de comunicación serie RS232 que puede ser fácilmente acoplado mediante la utilización de un circuito integrado del tipo MAX485, el cual es gestionado mediante los pines de control del XPORT. También existen sistemas embebidos del tipo XPORT que soportan varios estándares industriales [12] como Modbus, o inclusive el mismo RS485 directamente. Pero un sistema con la posibilidad de manejar los dos estándares genera un valor añadido bastante alto, tanto para la configuración como para la programación del sistema embebido XPORT y por ende para la gestión de la maquinaría.

En la interfaz implementada para la gestión de la maquinaria (ver figura 32) se ha añadido un selector que permite conmutar los estándares en comunicaciones RS232 y RS485. Esto se ha realizado con el fin de dejar un canal de configuración y gestión para el sistema embebido y a su vez para crear el puente entre el sistema de gestión y los módulos ICPDAS que forman parte del escenario de simulación industrial.

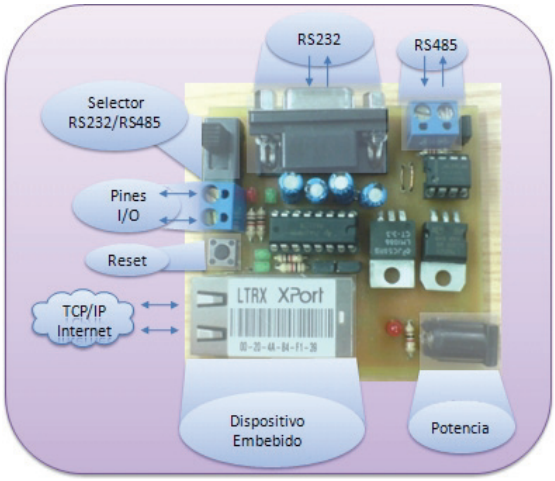

Fig. 32. Prototipo de la interfaz de gestión de la maquinaria

Para comprender el funcionamiento de este servicio es conveniente primero analizar el funcionamiento por defecto de la maquinaria. Esta maquinaria dispone de PLCs y máquinas simples (tornos, cintas transportadas, etc.). El dispositivo que se conectará a la maquinaria (en este caso el *XPort*) debe utilizar un determinado protocolo para comunicarse con los PLC. Este protocolo es bastante simple ya que todas las acciones se realizarán mediante dos comandos, uno de lectura de sensores y otro para actuar sobre los motores de las máquinas simples.

Como comando de lectura de sensores se enviará a los PLCs el comando "@0X" donde X se sustituye por el número del PLC cuyos sensores se quieren leer. Cuando se envía esto por el puerto serie, el PLC X responderá con una cadena hexadecimal con el valor de todos los sensores que él gestiona.

Como comando para actuar sobre los motores se enviará los PLCs el comando "@0XYYYY" donde X se sustituye por el número del PLC cuyos motor/es se quiere accionar e YYYY será una cadena hexadecimal correspondiente al estado de cada motor de dicho PLC. Por ejemplo se está trabajando con la maquinaria en estado de reposo y s la cinta transportadora 1 es gestionada por los bits 0 (Izquierda) y 1 (Derecha) del PLC 2, si se envía el comando @020001 la cinta se moverá hacia la izquierda.

4.1.2 Planteamiento de la gestión

Para gestionar el movimiento de las máquinas en un primer momento se almacenaron movimientos simples como por ejemplo, mover cinta transportadora 1 hacia la izquierda. Posteriormente se identificó que esta solución no era la acertada ya que con este mecanismo al accionar el motor de la cinta transportadora 1 apagábamos el resto de motores del PLC. Este problema provoca que el servicio tenga que conocer en todo momento el estado de la maquinaria, es decir, de todos y cada uno de los motores.

En paralelo a este problema de almacenamiento surge el problema de la seguridad en la maquinaria, ya que esta no implementa seguridad ninguna. Por ejemplo, si un torno llega al final de su recorrido no para automáticamente y sigue realizando su movimiento hasta romperse. Para resolver este problema el servicio deberá conocer que sensor indica la detención de un motor y enviar automáticamente el comando para detenerlo cuando el sensor se active. Esto plantea un problema triple ya que por un lado tenemos el problema de almacenar todos los sensores, el de refrescar con la suficiente rapidez estos y el de almacenar de alguna manera las reglas que dotan de seguridad al servicio.

Para implementar la propuesta se han planteado las tablas de la figura 33. Como se puede ver en la figura, se ha realizado una identificación directa entre el mundo real y las tablas, de manera que se almacenarán los PLCs y los sensores y motores de dicho PLC. Por otro lado, también se tiene una tabla que almacena las reglas de seguridad, relacionando un motor y un sensor, cuando el motor está realizando un determinado movimiento. Utilizando estas entidades el módulo es capaz de generar dinámicamente los comandos que necesita para comunicarse con los PLCs, refrescar el estado de todos y cada uno de los sensores y además establecer reglas de seguridad para detener los motores en caso de peligro. Para refrescar el estado de los sensores se han realizado varias pruebas y los mejores resultados se han obtenido para tiempos entorno a los 25ms.

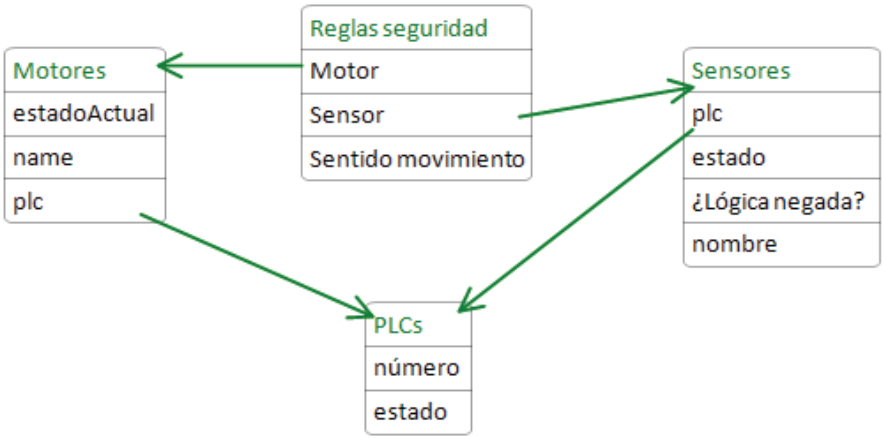

Fig. 33. Tablas utilizadas en el funcionamiento del servicio de la gestión de la maquinaria industrial

Para cargar toda la información en estas tablas se utiliza un archivo de configuración que tiene el aspecto que muestra la figura 34. En ella se puede ver cuatro partes en el archivo, una para cada tabla.

```
//PLCs (numero,bits dedicados a los motores, bits
dedicados a los sensores)
1#0#16
2#16#0
//Motores (Nombre,plc,bit dimension positiva,bit
dimension negativa) (El bit 16, estará fuera de rango e
indicará que el motor no se puede mover en esa dimensión)
M_CB1#2#0#1
M_CB2#2#2#3
M_TTCB#2#4#5
M_TTC#2#6#7
M_MTZ#2#8#9
M_MTD#2#10#16
//Sensores (Nombre,plc,bit,Logica negada o positiva[N|P])
S_CB1_1#1#0#P
S_CB1_2#1#1#P
S_CB2_1#1#2#P
S_CB2_2#1#3#P
S_TTCP#1#4#N
S_TTCN#1#5#N
S_TTCB#1#6#P
S_MTZP#1#7#N
S_MTZN#1#8#N
//Seguridad (Nombre motor, nombre sensor, estado
motor[N|P] (dirección del movimiento del motor)) -->
Cuando el motor este en el estado indicado y el sensor se
active, entonces el motor se parará.
M_TTC#S_TTCP#P
M_TTC#S_TTCN#N
M_MTZ#S_MTZP#P
M_MTZ#S_MTZN#N
//FIN
```

Fig. 34. Tablas utilizadas en el funcionamiento del servicio de la gestión de la maquinaria industrial.

Debido a este almacenamiento de tablas, el módulo podrá establecer reglas de seguridad dinámicas, de manera que se podrán realizar movimientos más complejos utilizando una sola invocación al servicio. Un ejemplo de estos movimientos sería “mover cinta transportadora a la izquierda hasta que se active sensor 7”. Para hacer esto se creará primero la regla de seguridad enlazando el motor de la cinta transportadora 1 y el sensor 7, y después se lanzará el comando para activar el motor en dicha dirección.

4.1.3 Dando forma a la interfaz del servicio

Una vez se sabe todo lo que es capaz de hacer el servicio sólo nos queda definir una interfaz adecuada. La elección de la interfaz es un tema bastante delicado y tras sopesar las distintas alternativas se seleccionó una interfaz homogénea, de manera que cuando se cambie de maquinaria, la interfaz siga siendo la misma y solamente

cambien los parámetros de los métodos (nombres de motores y nombres de sensores). La interfaz se compone de los siguientes métodos:

- *Arrancar*: Enciende un motor en una determinada dirección.
    - Recibe como parámetros el nombre del motor a arrancar y el sentido en el que debe hacerlo.
    - Retornará OK en caso de que todo vaya bien y ERROR en otro caso.
- *Detener*: Apaga un motor que está en movimiento.
    - Recibe como parámetros el nombre del motor a detener.
    - Retornará OK en caso de que todo vaya bien y ERROR en otro caso.
- *Sensores*: Obtiene el estado de los sensores de la maquinaria.
    - Retornará una estructura compleja con todos los sensores y sus estados.
- *Motores*: Obtienen el estado de los motores de la maquinaria.
    - Retornará una estructura compleja con todos los motores y sus estados.
- *RunYStop*: Este método enciende un motor en una dirección determinada y lo detendrá cuando un sensor se active.
    - Recibe como parámetros el nombre del motor a encender, el sentido en el que debe hacerlo y sensor que provocará la detención del motor.
    - Retornará OK en caso de que todo vaya bien y ERROR en otro caso.
- *PararTodo*: Este método supone apagar todos los motores de la maquinaria.
- *SetStop*: Este método se utiliza para establecer condiciones de parada, de manera que el motor que se pasa como parámetro se detendrá cuando se esté moviendo en la dirección indicada y se active el sensor indicado.
    - Recibe como parámetros el nombre del motor, el sentido y el sensor.
    - Retornará OK en caso de que todo vaya bien y ERROR en otro caso.

Con esta interfaz se puede ver como el único prerrequisito para que un usuario externo pueda utilizar los servicios es que conozca la topología de la maquinaria (motores, sensores y direcciones). Se podría haber definido la interfaz algo más ad-hoc a cada caso, pero eso hubiera requerido la compilación del código en cada maquinaria, perdiendo el carácter dinámico que hemos definido hasta el momento. Además en cualquier caso, por muy sencilla, dedicada y básica que se haga la interfaz, el usuario debe conocer la topología de la maquinaria para utilizar el servicio.

4.1.4 Utilizando el servicio de la maquinaria industrial

En esta sección se va a preparar un escenario de pruebas e invocar a los servicios de la interfaz para probar ver un ejemplo de su uso. En la figura 35 se puede ver la maquinaria industrial en estado de reposo y el escenario utilizado en el ejemplo. Nótese que la pieza se encuentra situada en CB3 y la intención será transportarla al otro lado de la maquinaria sobre el STS.

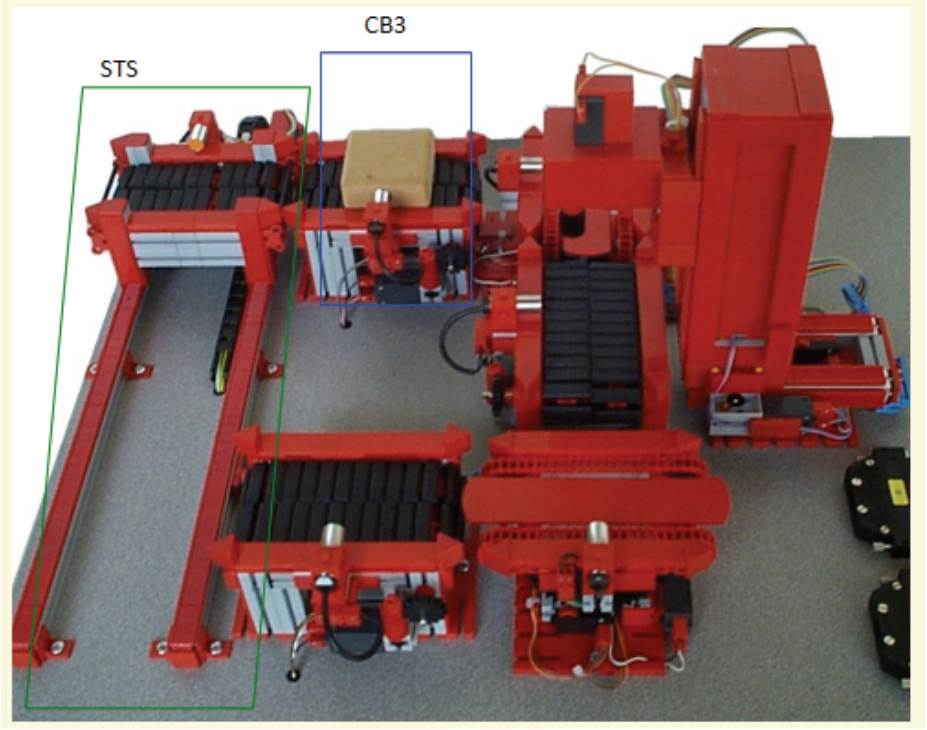

Fig. 35. Maquinaria industrial Staudinger GmbH en estado de reposo.

El primer paso que se realizará será ejecutar dos veces el método *runYStop*, la primera con los parámetros M_CB3, sentido negativo y sensor el que se encuentra situado sobre la STS, y la segunda con los mismos parámetros pero cambiando el motor, ya que en lugar de ser M_CB3, será el motor M_STSX, que mueve la cinta transportadora situada sobre el STS. Una vez se han ejecutado estos métodos la pieza avanzará hasta situarse sobre el STS, momento en el cual se detendrán las dos cintas. Tras situar la pieza sobre el STS sólo queda transportarla al otro extremo de la maquinaria. En este caso no será necesario utilizar la función *runYStop*, ya que existe una regla de seguridad definida para evitar que el STS se rompa al ir hacia abajo. Se utilizará pues el método arrancar y la dirección hacia abajo, esto provoca que el STS se desplace según los cálculos hasta el otro extremo de la maquinaria y se detenga frente a la otra CB.

4.2 Caso de uso: Gestión de redes

En esta aplicación se propone la confección de una sonda de red inteligente mediante el uso de dispositivos embebidos. La idea es incorporar el dispositivo embebido en la red que se quiere gestionar de forma que mediante el uso de una serie de servicios un sistema externo pueda gestionar los elementos que conforman la red gestionada.

Para este servicio se han adaptado las interfaces del sistema para adecuarse a la temática de redes. Accediendo a la IP del dispositivo (http://ipdispositivo) encontramos cuatro páginas, *home*, *interfaces*, *services* y *red*. Las tres primeras son muy similares a las ofrecidas en el módulo IMaaS, destacando por su nivel informativo *service*, y la página dedicada al servicio *red* (figura 36 y figura 37). En este caso la información mostrada depende de si hemos hecho uso o no de ciertos WS. Inicialmente mostrará una página sin información.

Fig. 36. Aspecto inicial de la página red.htm

Sin embargo, si se ha hecho uso de los WS de monitorización de red y se ha inicializado los servicios, la página mostrará información sobre el rastreo de equipos disponibles en nuestra red (figura 6), con la siguiente sintaxis:

xIP1.IP2.IP3-IP4-IP4….IP4

Donde IP1 es el primer byte de la IP, y los distintos IP4 son el último Byte, de modo que las IPs completas se formarían con las 3 primeras señaladas con la X y cada una de las IP4. Este modo de representación, lejos de razones de legibilidad, viene determinado por las limitaciones de espacio del dispositivo.

Fig. 37. Aspecto de la página red.htm una vez invocado el servicio de descubrimiento de equipos.

El total de equipos es el máximo teórico de la red, aplicando la máscara de red a la IP del *XPort*. En caso de redes muy pobladas, como por ejemplo una red/8 con todos los equipos levantados y respondiendo a comandos ping, se puede dar el caso de que no se pueda llevar control sobre todas las máquinas por las limitaciones de memoria.

4.2.1 Servicios

Se dispone de dos bloques de servicios, uno basado en la realización de comandos *ping*, otro basado en el *Simple Network Management Protocol* (SNMP).

4.2.1.1 Ping

Mediante esta utilidad (basada en el envío y recepción de paquetes ICMP) se puede controlar el estado de la conexión de los equipos de la red, es decir, conocer si un equipo está conectado o no. Notar que esto no asegura el correcto estado de la conexión de un equipo, puesto que el servicio puede estar deshabilitado. A continuación se describen los servicios desarrollados.

ping

Recibe la IP destino, en notación separada por puntos, y realiza una serie de pings para comprobar su conectividad (ver figura 38). Por defecto está configurado para realizar 10 intentos y con un retardo aceptado por petición de 30 ms. Si contesta con un 1 se podrá dar por válido el estado de la conexión.

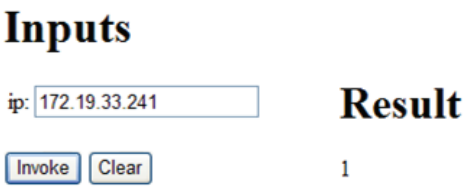

Fig. 38. Invocación y resultado de WS ping

refrescar

Se trata de un servicio sin parámetros, ni de entrada ni de salida, que realizará un barrido de la red del dispositivo en busca de equipos activos (que respondan a ping). El funcionamiento está basado en, partiendo de la IP del *XPort* y de su máscara de red, obtener la primera y últimas IP posibles de dicha red y realizar pings a todas ellas. Existe otro método de hacer ping a todos los equipos de una red, mediante el uso de la dirección de *broadcast*, pero produce sobrecargan en el dispositivo. Los parámetros de estos ping son más restrictivos que el del propio servicio ping, haciendo 2 peticiones y esperando tan solo 2 ms. El objetivo de esta limitación surge como respuesta a evitar una sobrecarga de la red, puesto que suelen ser muchas las direcciones posibles y un número elevado de peticiones por dirección podría causar pérdidas de rendimiento en la red. De los equipos que contestan nos guardamos su dirección IP en una lista, y ésta es precisamente la que se muestra en equiposred.htm.

Cada vez que se invoque este WS se recalculará el rango de IPs y cuáles son los equipos disponibles. Si se accede a la página Web mientras se está realizando este proceso, la página quedará cargando hasta que se complete, momento en que mostrará la información actualizada.

Puede ser útil en caso de desplazar el dispositivo de red o de restringirla/ampliarla mediante la máscara de red accediendo, como dijimos en primer lugar, mediante el puerto COM.

sigIP

Este servicio está orientado a su utilización por parte de una hoja de ejecución BPEL, concretamente para realizar algún tipo de acción sobre todos los equipos útiles de una red de forma secuencial, haciendo uso de un bucle.

Hace uso de la cadena que contiene todas las IPs de la red, proporcionando a cada llamada, por una parte, una IP de la lista (por orden), y por otra cuántas IPs quedan en la lista. Cuando llega a la última IP devolverá 0 en las IPs disponibles, y si se vuelve a realizar una nueva petición retornará números negativos en el número de IPs restantes y en el campo de la IP devolverá una secuencia de cuatro ceros (0.0.0.0).

En la figura 39 vemos el resultado de invocar sucesivamente al WS sobre una red de ejemplo.

Result returnp: quedan: 7 respuesta: 172.19.33.2	**Result** returnp: quedan: 6 respuesta: 172.19.33.11	**Result** returnp: quedan: 5 respuesta: 172.19.33.60
Result returnp: quedan: 4 respuesta: 172.19.33.86	**Result** returnp: quedan: 3 respuesta: 172.19.33.145	**Result** returnp: quedan: 2 respuesta: 172.19.33.162
Result returnp: quedan: 1 respuesta: 172.19.33.224	**Result** returnp: quedan: 0 respuesta: 172.19.33.241	**Result** returnp: quedan: -1 respuesta: 0.0.0.0

Fig. 39. Resultados de invocar consecutivamente el WS sigIP().

4.2.1.2 SNMP

El *Simple Network Management Protocol* (SNMP) es un protocolo de administración de dispositivos de red. En una red administrada dispondremos los dispositivos, los agentes que lleva cada dispositivo y los sistemas administradores de red (NMS). Los agentes poseen un árbol llamado MIB (Base de Información de Administración) con una estructura en la que almacenan información jerárquicamente. La forma de identificar cada objeto almacenado es por medio de los identificadores de objeto (OID), que representa escalarmente la ruta del árbol para llegar hasta ese objeto. Por ejemplo, 1.3.6.1.1 significaría acceder al objeto *directory* del árbol de la figura 40.

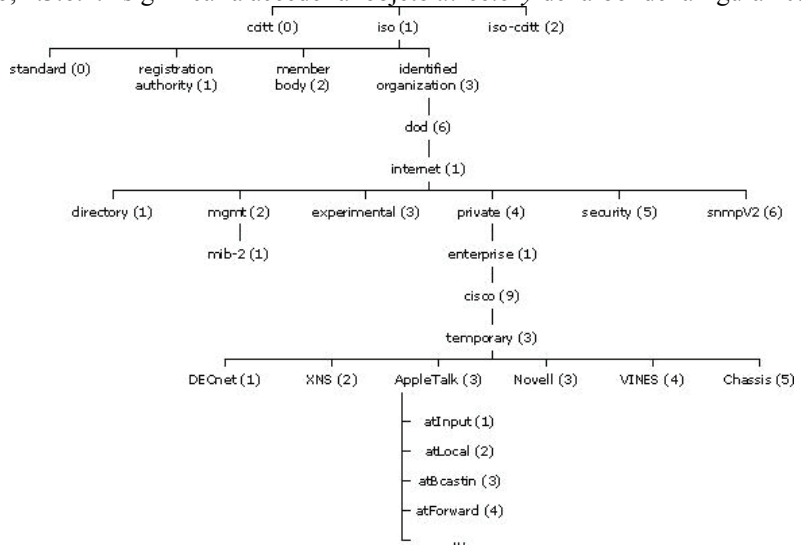

Fig. 40. Ejemplo de MIB

El dispositivo ha sido programado para trabajar como un NMS, enviando peticiones a los distintos agentes identificándolos por su IP.

Al estar el protocolo SNMP basado en UDP, nativamente no está soportado el envío y recepción síncronos de mensajes por lo que hay que establecer un estricto control sobre los mensajes generados. Para ello se han implementado varias políticas de seguridad. Por una parte cada paquete se estructura de una forma ordenada en la que en primer lugar se establece la longitud que se envía para que se sepa cuánto forma parte del paquete SNMP, puesto que no incorpora ningún carácter de terminación. Por otra parte, para controlar esta sincronía, cada mensaje establece un identificador de mensaje, cuya respuesta deberá tener el mismo identificador. Además, tras el envío UDP la aplicación espera un tiempo prudencial a recibir datos por el puerto UDP de SNMP (el 161 tanto para envío como para recepción). En caso de haber comenzado a recibir la respuesta (al conocer la longitud al comienzo del mensaje por la estructura de longitud + contenido comentada anteriormente) esperará hasta haberlo recibido por completo o bien pasado un tiempo excesivo se dará el error pertinente. Esto es así porque al ser consecuencia de la invocación de un WS, y esperar este una respuesta, hasta que no consiga la respuesta SNMP no se puede contestar al WS por lo que pueden darse problemas si se consiente un

comportamiento asíncrono. Encontraremos las dos opciones más necesarias de la administración mediante SNMP, el *get* y el *set*, es decir, solicitar información a los agentes por una parte y poder establecerla mediante el set.

La estructura de los paquetes SNMP (en el código se encuentra comentado Byte a Byte el significado de los mismos) la podemos ver en la figura 41.

La versión con que se trabaja de SNMP, la comunidad (medio de seguridad de esta versión) suele ser *public*, el tipo distingue de qué mensaje se trata (*get*, *set* o *response*), el error y su índice informan de disfunciones, el OID identifica el objeto al que queremos acceder bien sea para obtener información como para editarla, y por último en el valor estableceremos el tipo de dato a enviar y el dato que queremos, vacío para los gets.

Version	Comunidad	Tipo	Identificador	Estado de error	Índice de error	OID	Valor

Fig. 41. Paquete SNMP

Para las pruebas se puede instalar el agente SNMP de Windows (en nuestro caso XP) que viene como complemento en el CD de instalación.

A continuación se describen los servicios implementados.

getRequest

Con este servicio el NMS pide al agente de una IP dada el valor de un objeto que se especifica también como comando. Las comunidades del solicitante y del agente han de coincidir para que exista comunicación. El objeto específico se representa mediante su OID, mientras que la respuesta (si la hay) vendrá en el cuerpo del mensaje SNMP response que se obtenga del agente. En la figura 42 se muestra la invocación desde el entrono *Eclipse* solicitando el nombre de la máquina y el resultado obtenido. En la figura 43 se puede observar cómo el nombre corresponde, efectivamente con el del equipo.

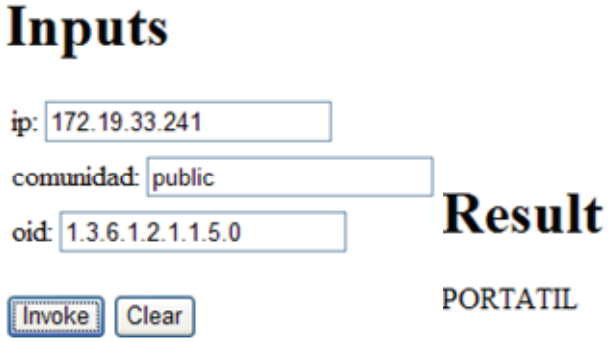

Fig. 42. Invocación y resultado del WS getRequest()

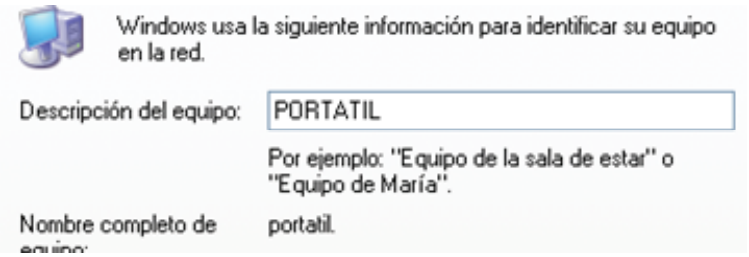

Fig. 43. Nombre del equipo mostrado desde MiPC

```
POST /redes HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.4
Host: 172.19.33.155
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 397

<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope xmlns:soapenv="http://
schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"><soapenv:Body><getRequest
xmlns="http://es.ua.dtic.redes"><ip>172.19.33.241</ip><comunidad>public</
comunidad><oid>1.3.6.1.2.1.1.5.0</oid></getRequest></soapenv:Body></
soapenv:Envelope>HTTP/1.0 200 Document follows
Content-type: text/xml

<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"><soapenv:Body>  <getRequestResponse xmlns="http://es.ua.dtic.redes">
   <respuesta>PORTATIL</respuesta>
  </getRequestResponse>
</soapenv:Body></soapenv:Envelope>
```

Fig.44. Contenido de peticiones SOAP del getRequest()

En la figura 44 se puede observar el cuerpo de las peticiones SOAP para el *getRequest* del ejemplo.

setRequest

En este caso el NMS envía una solicitud al agente para que modifique el valor del objeto identificado por el OID (figura 45). Además del valor, se ha de especificar el tipo de dato, a saber:

2 – Integer
4 – Octet String
5 – Null (si no enviamos dato)
6 – OID
40 – Dirección IP en notación con puntos
43 – Timeticks
47 – Unsigned Integer

Fig. 45. Invocación de setRequest() para establecer un nuevo nombre de equipo mediante SNMP

5 Conclusiones

En el presente trabajo se ha propuesto y desarrollado un framework de Servicios Web que permite la orquestación de los mismos. El framework ha sido concebido para su utilización en dispositivos embebidos con limitaciones de memoria y procesamiento.

Para su implementación se ha seleccionado el dispositivo *XPort* que ha puesto de manifiesto los problemas derivados de la falta de recursos en este tipo de dispositivos, haciendo patente la necesidad de realizar implementaciones restrictivas de los estándares asociados a los Servicios Web. En cualquier caso estas restricciones vendrían marcada por las capacidades del dispositivo en cuestión.

Se ha validado la propuesta en dos escenarios distintos, lo cual permite mostrar la flexibilidad del framework y su adaptabilidad a los cambios continuos de los requerimientos.

Como línea futura se propone la adaptación de la implementación del framework a otros dispositivos de carácter más general (p.e. los basados en *μClinux*).

Referencias

1. *Especificación WSDL del W3C*. 2001 [cited 2008 mayo 20]; Available from: http://www.w3.org/TR/wsdl.
2. *UDDI Data Structure Refrence*. 2002 [cited 2008 mayo 20]; Available from: http://uddi.org/pubs/DataStructure-V2.03-Published-20020719.htm.
3. *Especificación del estándar BPEL*. [cited 2008 mayo 20]; Available from: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

4. Newcomer, E., *Understanding Web Services – XML, WSDL, SOAP, and UDDI*. 2002: Addison Wesley.
5. *Especificación del XML*. [cited 2008 20 de Mayo]; Available from: http://www.w3.org/XML/.
6. *Using BPEL4WS in a UDDI registry* 2004 [cited 2008 20 de mayo]; Available from: http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-bpel.htm. .
7. *Using WSDL in a UDDI registry*. 2003 [cited 2008 mayo 20]; Available from: http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm. .
8. Lantronix. *Embedded Device Servers - XPort® - Embedded Ethernet Device Server* 2005 [cited 2008 mayo 20].
9. LANTRONIX, *XPort™ User Guide*. 2005, LANTRONIX.
10. *Staudinger GMBH*. [cited 2008 mayo 20]; Available from: http://www.staudinger-est.de/intl/en/simulation02.aspx?group=1.
11. ICPDAS. [cited 2008 Mayo 20]; Available from: http://www.icpdas.com/.
12. LANTRONIX. *XPort - Embedded Ethernet, Embedded Device Server, Serial To Ethernet*. [cited 2008 mayo 20]; Available from: http://www.lantronix.com/device-networking/embedded-device-servers/xport.html.
13. V. Gilart-Iglesias, F. Maciá-Pérez, D. Marcos-Jorquera and F. J.Mora-Gimeno. *Industrial Machines as a Service: Modelling industrial machinery processes*. Proceedings of 5th International IEEE Conference on Industrial Informatics (INDIN'07).Vienna, 2007.