Infraestructura de comunicaciones para algoritmos colaborativos

Universitat Oberta de Catalunya

Manuel Sopena Ballesteros

Ingeniería Superior de Informática

Proyecto Fin de Carrera

Dirigido por:

Dra. Eva Vallada Regalado

A mi padre cuya ilusión y ánimos

me han dado fuerzas para seguir adelante.

También hacer una especial mención a la tutora

del proyecto la Dra. Eva Vallada que ha sido

muy paciente conmigo, su ayuda ha sido de

muy gran ayuda, y espero que la solución comentada

en este proyecto le sea de ayuda en el futuro.

Indice

1. Introducción. 6
1.1 Justificación del PFC y contexto en el que se desarrolla. 6
1.2 Objetivos del PFC. 8
1.3 Enfoque y método seguido. 9
1.4 Planificación del proyecto. 9
1.5 Breve descripción de los otros capítulos de la memoria. 10
1.6 Herramientas de trabajo 11
1.7 Incidencias, riesgos y plan de contingencia 11
2 Estudio y elección de herramientas de trabajo 13
2.1 Investigación sobre las herramientas y metodologías de trabajo. 13
2.2 Criba de tres frameworks que mejor se adapten al proyecto. 14
2.2.1 MsgConnect: 15
2.2.2 Google Protocol Buffer 15
2.2.3 Windows Communication Foundation (WCF) 15
2.2.4 WWSAPI 16
2.3 Estudio y elección del mejor framework y breve descripción 18
3 Diseño del protocolo de comunicaciones. 20
3.1 Maestro-esclavo 21
3.2 Simulación de Islas 22
3.3 Serialización 24
4 Diseño del software 25
4.1 Servicio 25
4.2 Clientes 27
4.2.1 Wrapper 27
4.2.2 Librerías WWSAPI (ficheros wsdl y xsd) 38
4.2.3 Herramientas WWSAPI 38
4.3 Persistencia de datos 40
5 Implementación 41
5.1 Cliente 41
5.1.1 Modelo maestro-esclavo - Proxy 41
5.1.2 Modelo islas – Mensajes 42

5.2 Servicio 45
5.2.1 Modelo maestro-esclavo: 45
5.2.2 Modelo Islas: 46
5.2.3 DataContract 47
5.2.4 Hosting 47
5.2.5 Persistencia 47
6 Resultados y conclusiones 48
6.1 Pruebas locales 48
6.1.1 Pruebas modelo maestro-esclavo 48
6.1.2 Islas 54
7 Conclusiones 56
8 Posibles mejoras 57
9 Bibliografía 58

Figure 1 - Capas WWSAPI 17
Figure 2 - Casos de Uso 20
Figure 3 - Comunicacion maestro-esclavo 22
Figure 4 - comunicación islas....................23
Figure 5 - Funciones en el servicio maestro-esclavo....................26
Figure 6 - funciones en el servicio islas....................26
Figure 7 - funciones callback....................26
Figure 8 - Estructuras para serialización 27
Figure 9 - clase Wrapper....................28
Figure 10 - Clase ProxyIO Maestro-Esclavo 29
Figure 11 - enviar solución modelo maestro-esclavo....................32
Figure 12 - recibir solución modelo maestro-esclavo....................33
Figure 13 - Clase ProxyIO Islas 34
Figure 14 - Envío mensaje Islas....................37
Figure 15 - ejecutar servicio....................49
Figure 16 - comprobar servicio....................50

1. Introducción.

1.1 Justificación del PFC y contexto en el que se desarrolla.

En el mundo de los negocios y en concreto el proceso industrial está muy valorado el concepto Just-In-Time donde las tareas han de finalizarse acorde a una fecha de entrega sin demoras o adelantos. Demoras en los trabajos completados traen como resultado un cliente descontento, penalizaciones de contratos, pérdida de ventas y reputación, etc., pero un trabajo finalizado antes de tiempo puede acarrear gastos de inventario, devaluación del producto o pérdida de valor por deterioro.

El proceso industrial o programación de la producción consiste en la asignación de los recursos disponibles a una serie de procesos o trabajos durante un tiempo limitado. Es un proceso de toma de decisiones que intenta conseguir unos objetivos bajo unas restricciones de toda índole. Los recursos y tareas pueden formar parte de problemas muy diversos por ejemplo la asignación de robots en un proceso de producción en serie como la fabricación de automóviles, la asignación de pistas de aterrizaje en un aeropuerto a las aeronaves entrantes o salientes o asignación de herramientas a los trabajadores en un proceso de construcción de un edificio.

El problema de secuenciación de máquinas paralelas consiste en un conjunto $J = \{1, \dots, n\}$ de trabajos que se tienen que procesar en una serie de $M = \{1, \dots, m\}$ máquinas idénticas. Adicionalmente asumiremos las restricciones:

- Una máquina no puede procesar más de un trabajo al mismo tiempo
- Todos los trabajos y máquinas están disponibles en tiempo cero
- Las máquina nunca se dan de baja (siempre están disponibles)
- Un trabajo tarda el mismo tiempo en realizarse independientemente de la máquina que lo procese.
- A los trabajos se les pueden asignar distintas prioridades o pesos y cada tarea tendrá un tiempo de proceso distinto.

El objetivo de los problemas de secuenciación de máquinas paralelas es encontrar una secuencia *S* de trabajos que garanticen su resolución en la fecha de entrega. Cada trabajo tendrá un tiempo de procesamiento C_i. Sin embargo es posible que la fecha de finalización difiera de la fecha de entrega produciéndose una demora o anticipación en los tiempos acordados inicialmente, en estos casos se aplica una penalización si el trabajo es finalizado en un tiempo distinto al planificado. Esta planificación se denomina α_i en caso de terminar antes

(earliness) $E_i = \max\{0, (d - C_i)\}$ o β_i en caso de producirse una demora (tardiness) $T_i = \max\{0, (C_i - d)\}$.

El coste de una secuencia se puede dar como:

$$S = \sum_{i \in J} (\alpha_i Ei + \beta_i T_i)$$

La eficiencia de estos algoritmos se mide en 2 tipos:

- Minimizar el tiempo en que terminal la última tarea (minimización del makespan).
- Reducir al mínimo el número de tareas completadas después de la fecha de vencimiento.

En (VALLADA & RUIZ, 2007) se introduce el concepto de colaboración entre algoritmos, haciendo que las soluciones de las máquinas sean compartidas con las demás mejorando la eficiencia con respecto a algoritmos secuenciales. Es aquí donde empieza la propuesta de este proyecto.

La comunicación entre máquinas nos permite compartir soluciones y evolucionar el estado del sistema hacia una solución mejor, dotamos al sistema de una serie de herramientas que den a las máquinas la oportunidad de compartir sus soluciones mejorando la eficiencia del sistema. Cada máquina, analizará la solución que le llega del exterior (emigrante), y si es útil la tendrá en cuenta insertándole en su cola de inmigrantes. Actualmente se pueden diferenciar hasta cuatro tipos de comunicaciones distintas en la bibliografía:

Maestro-esclavo: el maestro distribuye los trabajos a los esclavos y les manda señales para que se sincronicen. En este caso se dice que la comunicación es síncrona ya que está controlada por el servidor.

Islas: cada máquina es independiente de las demás estableciéndose una comunicación asíncrona (no hay una autoridad que gestione las comunicaciones). Cada isla tiene independencia para trabajar en el mismo o distinto algoritmo que las demás.

Algoritmo cooperativo genético/ávido: este protocolo puede leerse en (VALLADA & RUIZ, 2007) y se considera una variante del método de islas al ser una comunicación asíncrona. Las máquinas han de decidir cuándo y qué soluciones compartir y con qué soluciones de las que le llegan se queda.

GRID: se usan recursos “ociosos” en una red como puede ser internet que ofrecen compartir sus recursos para la resolución de problemas.

Sin embargo hay que tener en cuenta qué emigrantes y cuando compartir sabiendo que esta información afectará a la evolución del algoritmo. La estrategia seguida es la de no

empezar a compartir candidatos hasta que el algoritmo este bien evolucionado de manera que entre todas las islas se abarque un espectro de soluciones lo suficientemente amplio (un envío demasiado pronto puede limitar la diversidad en las soluciones del sistema interrumpiendo en desarrollo individual de cada isla al dirigir sus acciones hacia un fin común). Normalmente se pospone el envío del primer emigrante hasta la mitad del tiempo total de proceso. Las soluciones candidatas a emigrantes han de pasar por un proceso de selección que garanticen su calidad antes de ser enviadas, los candidatos emigrantes han de proporcionar diversidad al sistema, uno de los procesos de selección se basa en el "mejor nueva solución" que selecciona aquel candidato que mejore la solución obtenida hasta el momento y sea nueva (no desarrollada hasta el momento).

1.2 Objetivos del PFC.

La motivación para el desarrollo de este proyecto de fin de carrera se centra en proporcionar un entorno de comunicaciones que permita a las distintas máquinas trabajar de forma conjunta y coordinada, compartiendo sus soluciones unas con otras, una vez se desarrolle el entorno pedido se espera obtener un sistema que mejore las soluciones ofrecidas anteriormente. Se ha demostrado que la comunicación tiene un factor determinante a la hora de mejorar los resultados obtenidos por métodos más tradicionales como los algoritmos secuenciales y se intentará mejorar las soluciones de los algoritmos obtenidos hasta ahora proporcionando esta nueva funcionalidad.

Este sistema de comunicaciones está basado en las investigaciones hechas por la Dra. Eva Vallada o en (VALLADA & RUIZ, 2007) y se integrará en problemas de secuenciación de máquinas paralelas o de la misma familia. Los algoritmos de las máquinas que resuelven este tipo de problemas ya están desarrollados por lo que sólo nos centraremos en el estudio de las herramientas disponibles para la definición del protocolo de comunicaciones y su implementación.

Se van a desarrollar dos tipos de comunicaciones distintas inspirados en los métodos encontrados en la bibliografía. El primero basara su modelo en el escenario maestro-esclavo y definirá una comunicación síncrona entre el cliente y el servicio en donde el cliente decida cuando enviar y recibir soluciones. Una segunda implementación del protocolo estará basada en el modelo de islas en donde se emulara un broadcasting donde el servidor enviará los datos recibidos a todos los clientes y además una comunicación asíncrona entre el servidor y el cliente de forma que el cliente no tenga que interrumpir sus tareas para recibir datos del servidor.

Una vez se tenga el sistema terminado, se pasará al entorno de pruebas para estudiar si el nuevo planteamiento mejora las soluciones obtenidas previamente. Se ensamblará este sistema con los algoritmos ya desarrollados y se pasará a la recolección de datos del banco de pruebas ejecutado. En las pruebas se ejecutarán los distintos algoritmos en diferentes

escenarios variando el número de máquinas y trabajo. Por último se calculará la media de todas las instancias ejecutadas en el mismo algoritmo.

1.3 Enfoque y método seguido.

Nos basaremos en un modelo cliente-servidor como modelo de desarrollo. La idea que perseguimos es abstraer el sistema de comunicaciones en los algoritmos de resolución de máquinas paralelas en un servicio externo de manera que las máquinas no necesitan desviar recursos para implementar el sistema de comunicaciones. Este servicio externo estará implementado por un servidor encargado de recibir y compartir las soluciones disponibles y además se podrá implementar distintas filosofías de comunicaciones para poder hacer las pruebas posteriores. Estas filosofías o modelos de comunicación estarán basados en los encontrados en la bibliografía (VALLADA & RUIZ, 2007).

Las filosofías elegidas a desarrollar son maestro-esclavo e islas que se detallarán en el capítulo Diseño del protocolo de comunicaciones.

Buscamos desarrollar un sistema de comunicaciones abierto y flexible, esto significa que nuestro servidor debería aceptar todo tipo de tecnologías en el cliente y queremos que nuestro sistema sea compatible con topologías de redes fácilmente escalables, pudiendo desplegarse tanto en redes locales como en redes WAN por medio de internet.

1.4 Planificación del proyecto.

Para la realización de este proyecto nos hemos basado en ocho etapas de trabajo:

Entendimiento del problema e identificación de objetivos:

- Conversaciones con el tutor del proyecto.
- Lectura de publicaciones en medios especializados.
- Entendimiento de las necesidades de las máquinas en algoritmos colaborativos.
- Entendimiento de los algoritmos y estructuras ya desarrolladas.
- Identificar las necesidades comunicativas de los algoritmos.

Diseño del protocolo de comunicaciones más adecuado para el problema:

- Establecer el entorno del problema.
- Acotar las funcionalidades a implementar.
- Documentar las funciones a implementar.

Estudio de los frameworks de comunicaciones y herramientas de desarrollo existentes:

- Estudio de las diferentes filosofías para la comunicación de algoritmos.
- Estudio de frameworks de comunicación.
- Catalogación y selección de candidatos más atractivos.
- Estudio de herramientas de desarrollo más adecuadas para implementar la solución.

Diseño y desarrollo de la aplicación a desarrollar:

- Estudio en profundidad el framework de comunicaciones.
- Identificar puntos débiles de las librerías y buscar alternativas.
- Diseño del comportamiento de la aplicación y del código.
- Desarrollo del programa.

Hosting y configuración de los entornos:

- Instalación de librerías y aplicaciones necesarias en las estaciones de trabajo.
- Configuración de la red de comunicaciones.

Testeo:

- Ejecución de banco de pruebas.
- Resolución de bugs.
- Desarrollo de mejoras.

Ejecución de banco de pruebas:

- Ejecución de pruebas en entornos de trabajo.
- Colección de datos.
- Análisis de datos.

1.5 Breve descripción de los otros capítulos de la memoria.

En primer lugar haremos un estudio de los requerimientos del sistema y plantearemos una solución que permita a las distintas máquinas compartir sus resultados parciales. Seleccionaremos las herramientas de trabajo para el desarrollo de las mismas que abarcan desde el entorno de desarrollo hasta las diferentes librerías que nos ayuden a desarrollar nuestro sistema de acuerdo a unos prerrequisitos. También comentaremos las limitaciones del framework escogido y cómo lo solventaremos.

A continuación procederemos con el diseño del protocolo de comunicaciones. Diseñaremos y explicaremos la topología de red y dos protocolos de comunicaciones para comunicar las máquinas con el servicio y mostraremos algunos diagramas que ayuden a su entendimiento.

Diseño del software: mostraremos y comentaremos los diagramas que ayuden a entender los algoritmos del sistema, además de las estructuras proporcionadas con la información de las soluciones proporcionadas por cada máquina.

Implementación: Comentaremos los puntos más interesantes sobre el desarrollo de la aplicación y su integración con el framework escogido.

Resultado y conclusiones: acoplaremos nuestro componente al ecosistema de resoluciones ya implementados y pasaremos al estudio de las soluciones obtenidas. Compararemos los resultados con los obtenidos anteriormente sin nuestro sistema de comunicaciones.

Posibles mejoras: Propondremos algunas variantes donde se pueda mejorar la solución propuesta o simplemente ampliar el estudio a otros protocolos de comunicaciones.

1.6 Herramientas de trabajo

El entorno en el que vamos a desarrollar este proyecto se basa en las siguientes características:

- Sistema operativo Windows 7 Professional 64 bits.
- Librerías .Net 4.5.
- Entorno de desarrollo Visual Studio 2010 Professional.
- Microsoft SDK v7.0 para las herramientas SvcTraceViewer para leer las trazas de WCF en el servidor, wstrace para obtener las trazas WWSAPI en el lado cliente, svcutil y wsutil para obtener y procesar los metadatos de WCF y crear las librerías de comunicaciones a bajo nivel.
- Office Professional 2012 para la realización de la memoria y presentación.

1.7 Incidencias, riesgos y plan de contingencia

Los problemas no previstos son un común en todo proyecto, por lo que deberemos crear un plan preventivo y correctivo para garantizar la continuidad del proyecto minimizando retrasos. Los errores más comunes en un proyecto de desarrollo software es la pérdida de la información o avería del equipo de trabajo.

Pérdida de información: se preparará un entorno de copias de seguridad en distintas localizaciones fuera del equipo de trabajo, se copiará la carpeta completa del proyecto Visual Studio con todos los ficheros fuentes y de configuración tanto del programa desarrollado como del IDE. Dentro de la carpeta se creará un archivo de texto con la descripción general de los

cambios respecto a la versión anterior. A parte de esto Visual Studio tiene su propio gestor de versiones que utilizaremos dentro de cada carpeta en caso de error manejando el código del proyecto.

Avería del equipo de trabajo: No disponemos de un equipo de repuesto, puesto que en caso de avería tendríamos que adquirir un equipo nuevo y reconstruir el software de nuevo.

2 Estudio y elección de herramientas de trabajo

2.1 Investigación sobre las herramientas y metodologías de trabajo.

Antes de empezar a programar necesitamos elegir nuestras herramientas de trabajo y la metodología a seguir. La primera duda a resolver es si desarrollaremos los componentes requeridos a bajo nivel con librerías socket o si de lo contrario haremos uso de herramientas a más alto nivel como frameworks o APIs que proporcionen las funcionalidades básicas. La mejor opción a priori es la de no "reinventar la rueda" y hacer uso de frameworks, comenzaremos haciendo un estudio sobre las distintas opciones disponibles, de entre todas las encontradas seleccionaremos tres o cuatro candidatos que serán objeto de una investigación en profundidad para comprobar cuál de ellos tiene la capacidad de producir lo que deseamos, y si no si son los suficiente flexibles como para adaptar la funcionalidad del producto a los requerimientos que describiremos a continuación.

Como se ha comentado anteriormente para desarrollar nuestro sistema de comunicaciones entre algoritmos que resuelven problemas de secuenciación de máquinas paralelas necesitamos librerías lo suficientemente flexibles que nos permitan serializar y de-serializar estructuras complejas entre los nodos de nuestra red. También se requiere flexibilidad a la hora de definir protocolos de comunicaciones personalizados que nos permitan definir comunicaciones síncronas y asíncronas además de que ambas partes han de ser capaces de empezar la comunicación independientemente y no sólo los clientes (no nos basta con el envío bidireccional de mensajes request/reply), nuestra idea es implementar diferentes tipos de comunicaciones entre el cliente y servidor para poder hacer comparaciones en las pruebas. Los algoritmos que se encargarán de comunicarse con nuestro servidor están implementados en C++ por lo que nuestras librerías deberían ser compatibles con este lenguaje. Escalabilidad es otra de los apartados en los que nos centraremos, sería bueno desarrollar un sistema escalable que de buen rendimiento tanto en redes LAN como en WAN basados en internet.

Serialización y de-serialización

La serialización consiste en el envío y recepción de datos entre el cliente y el servicio. Nuestro sistema debería permitir a las máquinas enviar sus estructuras de datos al servicio y viceversa. Además necesitamos que estos datos sean entendidos y procesados por ambas partes, incluso si hacen uso de tecnologías distintas.

Flexibilidad a la hora de definir las comunicaciones

Nuestro sistema ha de ofrecer un medio de comunicación bidireccional entre el cliente y el servicio de manera que cualquiera de ellos pueda iniciar una conversación. Para el modelo maestro-esclavo necesitamos una comunicación request/reply, es decir el cliente ha de invocar una función en el servidor y éste enviar el resultado de la función de vuelta. Otro caso es el modelo de islas en donde el servidor ha de hacer un broadcasting al resto de clientes, esto implica que el servidor ha de capaz de consumir funciones en el cliente.

Los clientes se basan en Visual C++

Las librerías con las que desarrollaremos nuestra aplicación deben ser compatibles con la tecnología cliente en C++. Además es deseable que podamos usar clientes desarrollados en otras tecnologías.

Persistencia de datos en el servidor

Los datos enviados en el servicio han de ser permanentes durante la ejecución de los algoritmos de ejecución.

2.2 Criba de tres frameworks que mejor se adapten al proyecto.

Frameworks de comunicaciones y Serialización de datos hay muchos pero tenemos que tener en cuenta que la tecnología usada en nuestros clientes es Visual C++ (.Net). Además la comunicación que perseguimos no se basa en la típica request/reply sino que también buscamos implementar un sistema de broadcasting obligándonos a implementar llamadas desde el servidor al cliente de forma asíncrona.

Para nuestro estudio seleccionaremos cuatro frameworks de desarrollo:

- MsgConnect
- Google Protocol Buffer
- Windows Communication Foundation
- WWSAPI

2.2.1 MsgConnect:

Es un conjunto de librerías de alto nivel bastante completo diseñado por Eldos Corporation que ofrece todo lo que necesitamos para desarrollar el sistema de comunicaciones. La filosofía detrás es la de emular Windows messaging trabajando con mensajes y colas siendo los primeros las estructuras que contienen los datos a enviar y los segundos se configuran para enviar y recibir mensajes. MsgConnect también nos brinda la posibilidad de desarrollar nuestro propio protocolo de comunicaciones modificando el comportamiento de las colas, se pueden hacer programación con hebras para comunicaciones asíncronas, también soporta la posibilidad de hacer broadcasts y callbacks para llamar funciones en el cliente desde el servidor y proporciona mecanismos de seguridad. Se eligió para el estudio porque ofrece todo lo que necesitamos para desarrollar el sistema que requerimos.

2.2.2 Google Protocol Buffer

Es un lenguaje propio, independiente de la plataforma y extensible para serializar estructuras de datos. Google protocol buffer no ofrece capa de comunicación por lo que nos obligaría a desarrollar nuestra capa de comunicaciones usando librerías a bajo nivel con sockets tanto en la parte cliente como en el servidor. La versión oficial sólo trabaja con tecnologías Java, C++ y pythom, pero existe una versión propietaria y gratuita llamada protobuf para tecnologías .Net. Con Google Protocol Buffer podemos implementar la lógica que codifica los datos que viajan por el canal de datos de manera que el destinatario pueda entenderlos.

2.2.3 Windows Communication Foundation (WCF)

WCF ofrece un conjunto de librerías para el desarrollo de aplicaciones orientadas a servicios, ofreciendo la posibilidad de enviar datos asíncronos entre dos aplicaciones remotas, los mensajes pueden ser tan diversos como caracteres en un fichero XML como secuencias de datos binarios.

Entre las características de WCF podemos encontrar:

- **Orientado a servicios:** WCF se basa en la filosofía SOA (Service Oriented Architecture) que consiste en la transmisión de datos mediante servicios web. Este tipo de aplicaciones tienen un acoplamiento muy reducido permitiendo a clientes de cualquier tecnología comunicarse con nuestro servicio.
- **Interoperabilidad:** WCF es compatible con multitud de tecnologías entre las que destacamos protocolos web services (HTTP, SOAP, XML, etc.), aplicaciones COM, COM+, etc.
- **Varios modelos de mensajes:** WCF ofrece la posibilidad de enviar mensajes solicitud/respuesta, mensajes unidireccionales (sólo uno de los extremos envía un mensaje sin esperar la respuesta) o dúplex donde ambos extremos establecen una conexión y pueden enviar mensajes independientemente.

- **Metadatos de servicios:** WCF ofrece la posibilidad de publicar los metadatos de los servicios mediante estándares como WSDL y XSD.
- **Contratos de datos:** Permite la creación de contratos sencillos mediante clases .Net. Básicamente se pueden definir clases en C# que implementen los datos y la lógica que se quieren transmitir.
- **Seguridad:** se pueden configurar parámetros de seguridad que obliguen a los clientes a autentificarse o también se pueden cifrar los mensajes enviados.
- **Varios transportes y codificaciones:** se pueden usar diferentes tipos de transporte de datos y codificación como mensajes SOAP sobre HTTP o envío de mensajes en TCP, etc.
- **Fiabilidad en el envío de los mensajes:** WCF garantiza el envío de los mensajes si el destinatario no está disponible e incluso si se producen errores en el transporte de datos.
- **Mensajes duraderos:** se garantiza la persistencia de los mensajes mediante bases de datos que garantizan su envío incluso si el destinatario no está disponible.
- **Transacción:** las transacciones permite definir un grupo de acciones dentro de una transacción atómica, garantizando la correcta ejecución de todas sus instrucciones y que requiera una confirmación en su ejecución, de otra manera no se podrán ejecutar las instrucciones.
- **Compatibilidad con AJAX y REST.**
- **Extensibilidad:** posibilidad de customizar determinados servicios.

Estas librerías están integradas en todo sistema Windows tanto en entornos de escritorio como servidor y abarca todos los apartados de comunicación que necesitamos, desde la capa de transporte de datos (serialización), pasando por las comunicaciones (publicación y consumo de servicios) y persistencia (datos accesibles en todo momento).

2.2.4 Windows Web Services API (WWSAPI)

Es una API para desarrollar aplicaciones SOAP basadas en web services. WWSAPI se basa en C/C++ y ofrece serialización mediante mensajes SOAP que se basa en XML garantizando compatibilidad con multitud de tecnologías y la distribución de sus librerías en capas facilita la creación de protocolos personalizados:

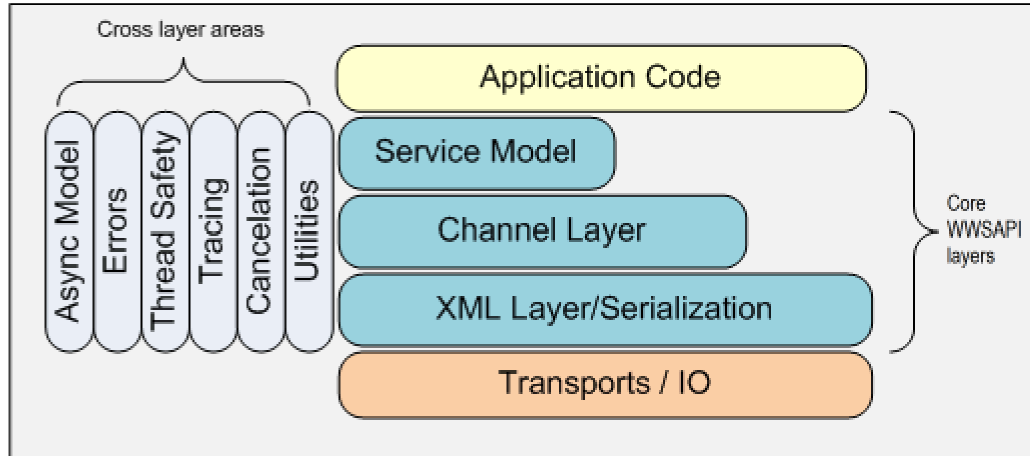


Figure 1 - Capas WWSAPI

- **Capa de Servicios:** Es la capa de alto nivel que ofrece un modelo de programación basada en métodos, es la capa más sencilla de usar. Ofrece la parte Host para programar en el lado servidor y la capa Proxy para programar en el lado cliente.

- **Capa de Canales:** gestión de mensajes que contienen los datos a enviar, canales de comunicaciones con los que se usa para enviar y recibir canales entre dos endpoints. Esta capa es más complicada de usar pero en cambio ofrece una gran flexibilidad para crear protocolos de comunicaciones personalizados.

- **Capa XML:** Ofrece las funciones para crear y leer contenidos de mensajes en XML para serializar y de-serializar datos.

2.3 Estudio y elección del mejor framework y breve descripción.

	MsgConnect	**Google Protocol Buffer**	**Windows Communication Foundation + Windows Web Services API**
Ventajas	• Poco acoplado, por lo que se podría reutilizar el algoritmo en distintos entornos.	• Muy testeado y optimizado. • Serialización de objetos. • Buen soporte con una comunidad detrás muy grande.	• Muy testeado y optimizado. • Serialización de objetos integrada. • Soporte nativo en sistemas Windows. • Minimización de dependencias entre cliente y servidor. • Tiempo reducido a la hora de arrancar los servicios • Muy buen soporte con una comunidad detrás muy grande. • Ofrece lógica para establecer sesiones. • Soporta sistema de comunicaciones dúplex con callback que permiten al servidor hacer peticiones al cliente.
Desventajas	• Pago • Comunidad pequeña (nos vemos restringidos a llamar al soporte del framework en caso de problemas). • Código a alto nivel, no ofrece la posibilidad de aprender tecnologías de referencia como web services o SOAP	• No ofrece capa de red para establecer comunicaciones entre maquinas por lo que la lógica de sesión y transporte de datos debería implementarse.	• WCF es una tecnología más enfocada a C#.

Windows Communication Foundation se posiciona como la mejor opción al ser una plataforma Microsoft y sus librerías están integradas en el sistema operativo Windows (escritorio y servidor) ofreciendo un buen rendimiento en los entornos de pruebas. WCF ofrece todo lo que necesitamos en una única tecnología (capa de Serialización, comunicaciones y

persistencia) y además ofrece distintas capas de desarrollo de alto a más bajo nivel permitiendo crear entornos más flexibles acorde con las necesidades del programador. Respecto a la documentación y soporte, WCF Tiene una comunidad de desarrolladores muy grande siendo muy fácil encontrar documentación y foros de desarrollo. Uno de los problemas que tiene WCF es que está enfocado a C# y las herramienta de desarrollo oficial (Visual Studio 2012) no ofrece integración con clientes Visual C++, esto puede solucionarse con dos alternativas; C++\CLI o WWSAPI. C++\CLI es una versión modificada de Visual C++ o managed C++ al que se le han añadido nuevas expresiones gramaticales y sintácticas con el objetivo de integrar C++ con .NET, dentro de los inconvenientes en adoptar C++/CLI podríamos nombrar que perderíamos la flexibilidad multiplataforma que queremos darle al proyecto, otro de los inconvenientes que encontramos es que no existe mucha documentación para su integración con WCF. Una segunda opción que se propone como más adecuada es hacer uso de las librerías Windows web services de Microsoft (WWSAPI). Las ventajas de usar WWSAPI sobre C++\CLI radican en que podemos reutilizar el código de las máquinas ya implementado haciendo uso de librerías WWSAPI de comunicación, WWSAPI al igual que WCF ofrece distintos niveles de programación, el nivel más alto llamado capa de servicios se basa en proxis y es independiente de la tecnología cliente. Dado esto y sabiendo que el proyecto actual de algoritmos colaborativos se está migrando a C#, WCF se posiciona como una buena solución una vez solventado el problema de comunicación con C++. El personal implicado en la migración a C# podría hacer uso de este proyecto haciendo sencillas modificaciones en el código.

Google protocol buffer en su contra no ofrece una plataforma de comunicación basándose sólo en la Serialización de datos para ser transportados. Uno de los factores más importantes de Google protocol Buffer es su portabilidad ya que puede ser usado en cualquier infraestructura, pero en nuestro caso nos vamos a basar en tecnologías Windows/.Net por lo que esto no es un factor decisivo.

Msgconnect ofrece un sistema de comunicación y serialización multiplataforma, ofreciendo lo que necesitamos para el desarrollo del proyecto pero en su contra es una tecnología de pago y su rendimiento está por debajo de WCF ya que no es una aplicación nativa en el SO a utilizar. Otro punto en contra es que su comunidad es pequeña limitándose al soporte que ofrece la compañía detrás y un pequeño foro en su página oficial.

3 Diseño del protocolo de comunicaciones.

Implementaremos nuestro protocolo a partir de un diseño básico de comunicaciones bidireccional en donde un cliente enviará sus estructuras al servicio, el servicio hará de hub y compartirá las estructuras con el resto de clientes. En el siguiente diagrama de casos de uso se puede ver la funcionalidad básica.

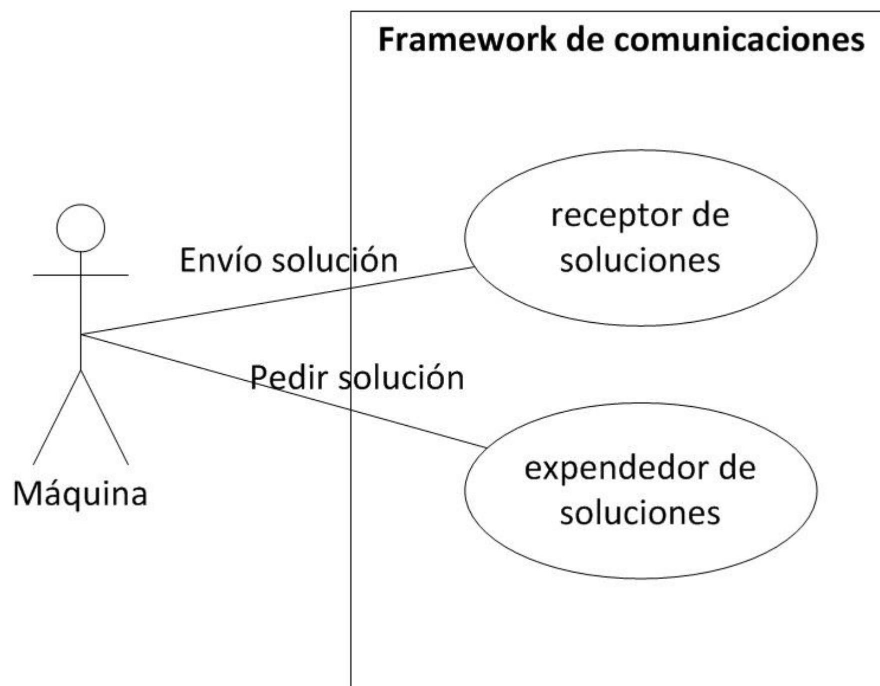


Figure 2 - Casos de Uso

A partir de este diseño desarrollaremos dos versiones distintas del protocolo basadas en la bibliografía e introducidos anteriormente. La primera es una versión del modelo maestro-esclavo y la segunda del modelo de islas, cada uno de estos protocolos ofrecerá las siguientes funcionalidades:

Protocolo maestro-esclavo:

1. El cliente envía sus estructuras al servidor.
2. El servidor almacena las estructuras recibidas durante todo el proceso de ejecución del algoritmo.

3. El cliente solicita las “n” mejores estructuras creadas hasta el momento.
4. El Servidor envía las “n” mejores estructuras recibidas al cliente que las solicitó.

Protocolo basado en islas:

1. El cliente envía su estructura al servidor.
2. El servidor envía la estructura recibida al resto de clientes.

3.1 Maestro-esclavo

En este modelo el cliente decide cuando enviar y recibir soluciones al servidor, el servidor básicamente ofrecerá dos métodos, uno para recibir datos del cliente y otro para enviar las “n” mejores estructuras al cliente que lo solicite. Decimos que esta comunicación es síncrona porque el cliente siempre sabe cuándo va a recibir datos puesto que es él quien empieza la comunicación. La idea es que el servidor vaya almacenando las soluciones recibidas en una lista de manera que cuando un cliente solicita un conjunto de soluciones, el servidor envía una lista con las mejores soluciones recibidas de entre todos los clientes. La funcionalidad en la parte del cliente se implementará usando la capa de servicio WWSAPI que nos proporciona los métodos para trabajar con proxis. Los métodos WWSAPI y la implementación del proxy se crearán a partir de herramientas svcutil y wsutil. Svctuil se encargara de extraer los metadatos en ficheros wsdl y xsd que describen la estructura de datos a transmitir y las funciones publicadas por el servicio, en cambio, Wsutil se carga de implementar el código para manejar las estructuras y funciones en los ficheros wsdl y xsd. El protocolo elegido para comunicar el servicio con los clientes es HTTP. Además los datos enviados al servicio han de ser persistentes durante todo el proceso de ejecución, esto significa que nuestro servidor ha de mantener los datos que recibe en todo momento durante la ejecución del algoritmo.

Escenario:

1. Cliente envía solución al servicio.
2. Servicio almacena la solución en una lista y las ordena de mejor a menor.
3. Cliente solicita “n” mejores soluciones.
4. Servicio envía “n” mejores soluciones a los clientes.

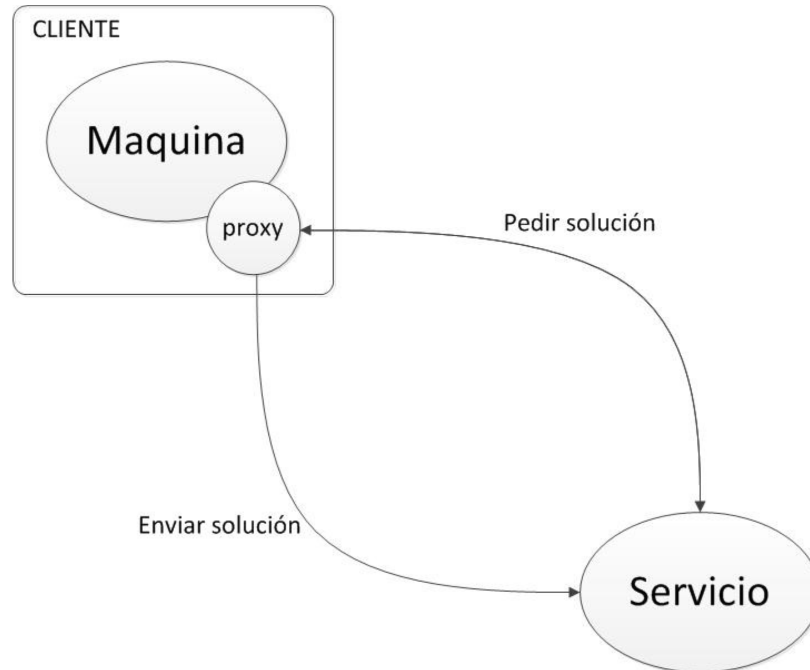


Figure 3 - Comunicacion maestro-esclavo

3.2 Simulación de Islas

En este modelo el cliente controla el envío de datos pero no la recepción, es decir, una vez un cliente envía sus datos al servidor, este automáticamente los reenvía al resto de clientes. Este tipo de comunicación se llama asíncrona puesto que el cliente nunca sabe cuándo va a recibir datos y por lo tanto no tiene que parar su flujo de ejecución para solicitar nuevos datos al servidor. Esta funcionalidad añade una nueva complejidad en la comunicación puesto que el servidor necesita empezar un nuevo flujo de comunicaciones para enviar los datos al cliente, esto implica que el cliente haga parte de servidor proporcionando una función para este cometido, por lo tanto podemos decir que las comunicaciones pueden comenzar tanto por parte del cliente como del servicio, a esto se le llaman comunicaciones dúplex y necesitan que la tecnología en el cliente lo soporte. Otro de los problemas que aparecen con este tipo de comunicaciones es como decirle al servicio a qué clientes enviarles las estructuras, para ello necesitamos definir un sub-protocolo para para dar de alta y de baja a los clientes en el servicio de forma que este tenga un registro de los clientes a los que enviar datos. Para dar de alta un cliente, el servidor procede almacenando la información del canal de datos del ciente para llamar a sus funciones. Este proceso se llama callback y WWSAPI no lo soporta, por lo que en principio no se podría usar los canales dúplex proporcionados por. esto lo solucionaremos creando un canal TCP socket en WWSAPI que usa comunicación dúplex de forma nativa y definiremos el mismo canal en el servicio WCF. Además debemos definir un listener en el cliente que recoja los elementos que envía el servicio de forma asíncrona, esta es la principal razón por la que no podremos trabajar directamente con proxis de la capa de

servicios de WWSAPI y tendremos que utilizar las funciones de bajo nivel de la capa de canal de datos para crear mensajes y canales de datos directamente.

Escenario:

1. Cliente se da de alta en el servidor
2. Cliente envía solución al servicio.
3. Servicio recibe los datos y los reenvía a todos los clientes subscritos por medio de broadcasting.
4. Cliente se da de baja en el servidor cuando acaba su ejecución.

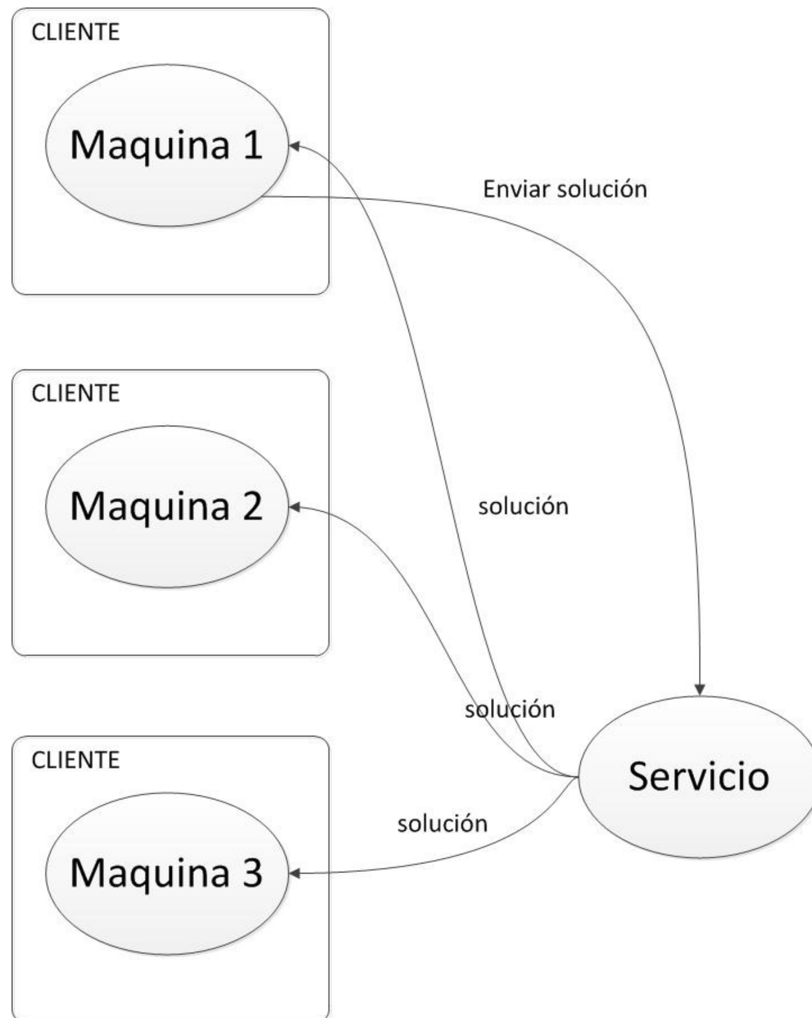


Figure 4 - comunicación islas

3.3 Serialización

La serialización se encarga de transformar los datos a un formato adecuado para que puedan ser enviados por el canal de datos hasta el destinatario. En nuestro caso el cliente no puede enviar los datos en el mismo formato con el que trabaja internamente puesto que el servicio no lo entendería al ser tecnologías distintas (esto es una de las consecuencias de hacer la tecnología en el servicio compatible con terceros), lo mismo pasa si aplicamos esta lógica a la inversa, por eso necesitamos usar una estructura intermedia que puedan entender ambas partes, esta estructura será la encargada de viajar entre el proxy en el cliente y el stub en el servicio, además, aparte de definir estas estructuras necesitaremos crear toda la lógica alrededor que nos permita traducir estos datos al formato adecuado que entiendan el cliente y el servicio, estas funciones las llamaremos envolturas o wrappers.

4 Diseño del software

En este apartado se va a explicar las particularidades encontradas a la hora de programas las diferentes partes de nuestro algoritmo.

Antes de entrar en detalle comentaremos algunos términos de WCF que se usarán más adelante:

En WCF la comunicación se establece entre endpoints, los endpoints proporcionan acceso a las funcionalidades del cliente.

Un endpoint está formado por:

- Dirección: que indica dónde puede encontrar el endpoint
- Binding: que especifica cómo un cliente puede comunicarse con el endponit
- Un contrato: que especifica las operaciones disponibles
- Un conjunto de comportamientos: que especifican detalles del endpoint

4.1 Servicio

Para el servicio vamos a crear una aplicación puramente WCF, esto significa que el código estará escrito en C# con todas las ventajas que esto implica. El servicio está compuesto por dos tipos de interfaces junto con su implementación y las clases de datos intermedias que se utilizarán en la serialización (SolucionGSer). Un tipo de interfaces definirán las operaciones en el servicio (contrato) que pueden ser consumidas desde los clientes y las otras las operaciones en el cliente que pueden ser llamadas desde el servicio (callback). Los contratos se definirán para los dos modelos (maestro-esclavo e islas), mientras que los callbacks sólo serán necesarios implementarlos en el modelo de islas.

A continuación se pueden consultar las definiciones de las interfaces.

ICMPFSP_Service_MS implementa las funciones del contrato en el modelo maestro-esclavo.

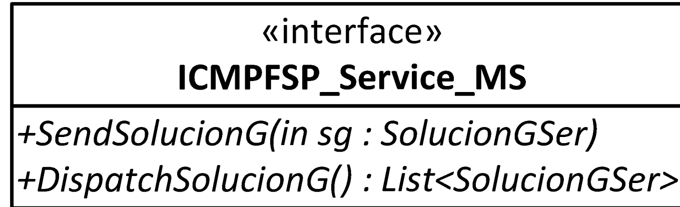


Figure 5 - Funciones en el servicio maestro-esclavo

ICMPFSP_Service_Islas implementa las funciones en el contrato para el modelo de Islas.

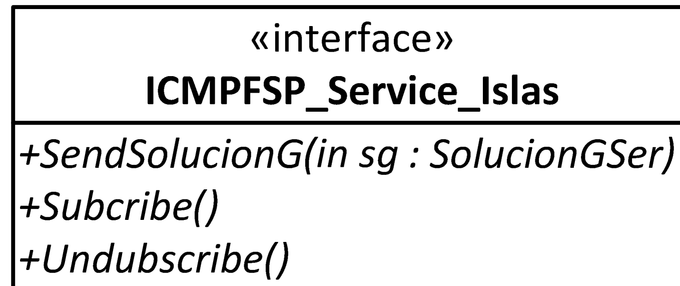


Figure 6 - funciones en el servicio islas

ICMPFSP_Service_Islas_Callback define las funciones callback en los clientes que puede consumir el servicio.

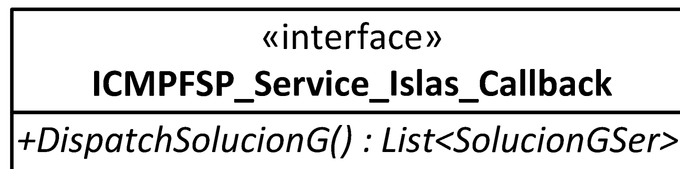


Figure 7 - funciones callback

En los diagramas siguientes se puede consultar las estructuras intermedias para la serialización de datos:

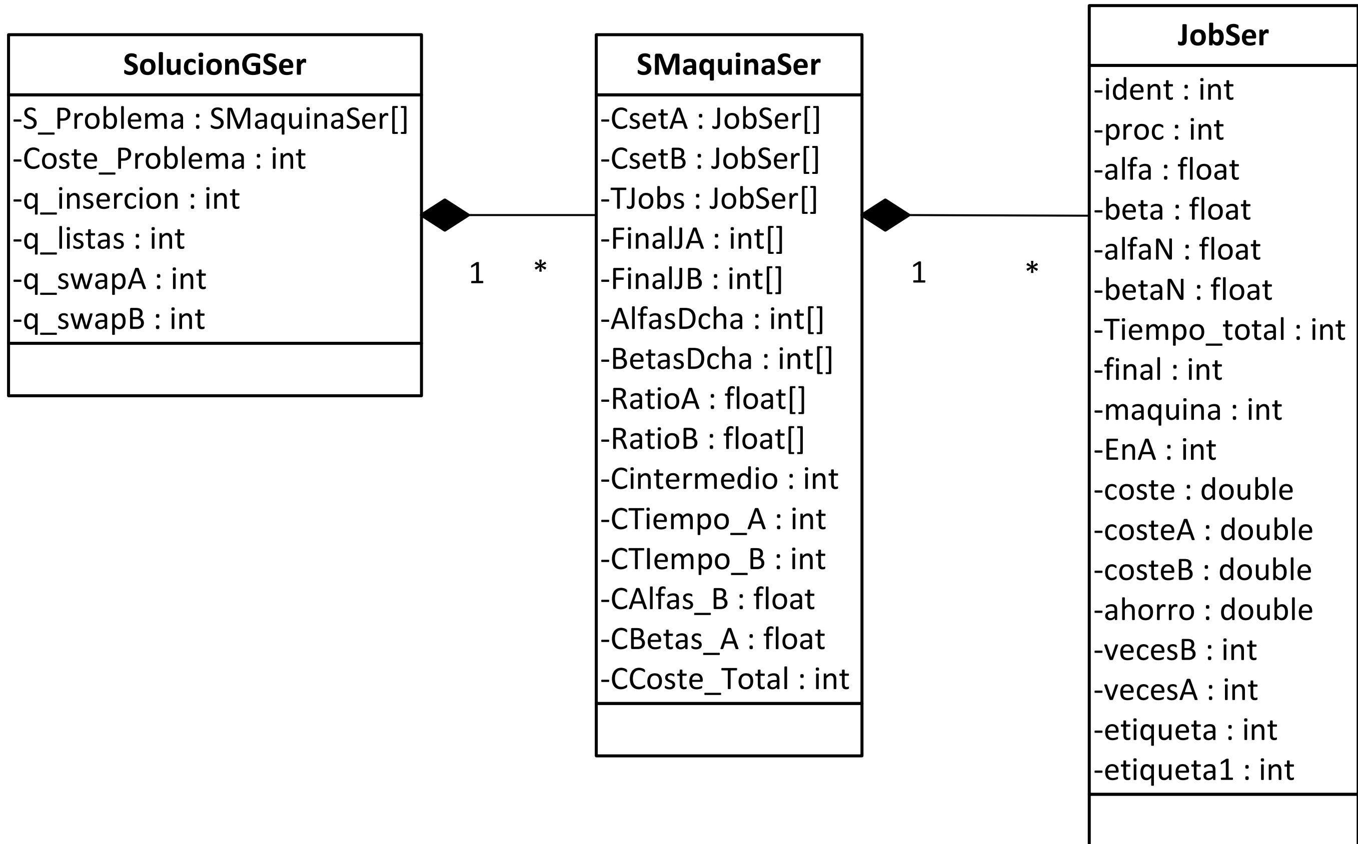


Figure 8 - Estructuras para serialización

4.2 Clientes

Nuestros clientes necesitan de unas librerías intermedias que les permitan consumir, publicar y serializar los datos de sus soluciones locales de forma fácil minimizando los cambios en el código de las máquinas. Las librerías WWSAPI son complejas de usar por lo que implementaremos una API que proporcione a los clientes las funciones básicas para la parte comunicativa. Esta API se dividirá en dos ficheros Wrapper y ProxyIO. Estas librerías simplificarán el uso de WWSAPI para la comunicación entre el cliente y el servicio inicializando y configurando las estructuras necesarias.

4.2.1 Wrapper

Esta librería se encargará de transformar los datos originales en el cliente C++ (SolucionG, SMaquina y Job) en las estructuras intermedias (SolucionGSer, SMaquinaSer y JobSer) que entiendan el proxy WWSAPI y el stub en el servicio. WWSAPI sólo entiende tipos simples y estructuras de C, por lo que tenemos que adaptar las estructuras en el cliente migrando las listas y vectores a arrays, el resto de datos se mantendrán igual.

Las funciones a implementadas serán:

- JobW: devuelve la estructura JobSer equivalente a los datos Job enviados.
- JobUW: devuelve la estructura Job equivalente a los datos JobSer enviados.
- SMaquinaW: devuelve la estructura SMaquinaSer equivalente a los datos SMaquina enviados.
- SMaquinaUW: devuelve la estructura SMaquina equivalente a los datos SMaquinaSer enviados
- SolucionGW: devuelve la estructura SolucionGSer equivalente a los datos SolucionG enviados.
- SolucionGUW: devuelve la estructura SolucionGSer equivalente a los datos SolucionG enviados.

A continuación se puede consultar la definición formal de la clase Wrapper.

Wrapper
+static JobW(in j : Job) : JobSer* +static JobUW(in j : &JobSer) : Job* +static SMaquinaW(in sm : SMaquina) : SMaquinaSer* +static SMaquinaUW(in sms : &SMaquinaSer) : SMaquina* +static SolucionGW(in sg : SolucionG) : SolucionGSer* +static SolucionUW(in sms : &SolucionGSer) : SolucionG*

Figure 9 - clase Wrapper

ProxyIO

Esta librería implementa la lógica de envío y recibimiento de datos que las máquinas/clientes usan para gestionar sus comunicaciones con el servidor. El código de estas funciones difieren dependiendo del tipo de modelo (maestro-esclavo e islas) en cuestión, cada modelo hará uso de una capa WWSAPI distinta dependiendo de la flexibilidad que necesitemos para desarrollar nuestra funcionalidad. Como ya se comentó antes, haremos uso de dos capas WWSAPI, una de alto nivel llamada capa de servicios (usada en el modelo maestro-esclavo) para gestionar los proxis y otra a más bajo nivel (usada en el modelo islas) que gestiona los canales de datos y mensajes directamente dándonos más flexibilidad a la hora de definir las características de nuestro protocolo.

Entre las tareas que gestiona esta librería podemos encontrar:

- Gestión del heap: el heap es la parte de memoria que el proxy usa para la serialización de datos, cuando definimos un proxy le asignamos el tamaño de la memoria que ha de ser lo suficientemente grande como para contener el flujo de datos recibidos del servidor. La gestión del heap es común en la capa de servicios y en la de mensajes.
- Gestión del proxy: el proxy es la estructura a alto nivel que se encarga de enviar datos al servicio, al trabajar con la capa WWSAPI de servicios implica que tendremos a nuestra disposición las funciones en el contrato disponibles para su llamada, estas funciones necesitan como parámetro nuestro proxy con la información del endpoint y los datos de las soluciones locales serializadas.
- Gestión de mensajes: en WWSAPI los mensajes son objetos que encapsulan datos para ser enviados o recibidos. La gestión de mensajes es parte de la capa de mensajes de WWSAPI.
- Gestión del canal de datos: el canal de datos se encarga de establecer una comunicación entre dos o más nodos y se usan para enviar y recibir mensajes. La gestión de canales de datos forma parte de la capa de mensajes WWSAPI.

4.2.1.1 Maestro-esclavo

Este modelo implementará cuatro funciones, ProxyIO, Enviar, Recibir y ErrorProcess.

- ProxyIO: es el constructor que construye e inicializa las estructuras necesarias para la comunicación.
- Enviar: envía una estructura SolucionG al servicio.
- Recibir: solicita al servicio un vector con las mejores SolucionG enviadas por los clientes.ErrorProces es un método interno para gestión de errores.

A continuación se puede consultar la definición formal de ProxyIO para el modelo maestro-esclavo.

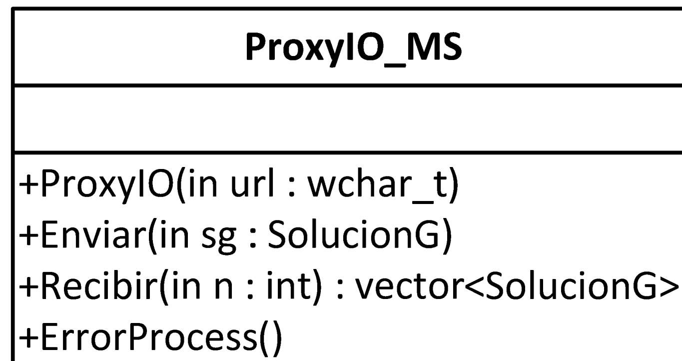


Figure 10 - Clase ProxyIO Maestro-Esclavo

Funciones WWSAPI utilizadas en este modelo:

WsCreateHeap: crea el heap que gestiona la memoria para serialización de datos.

```
HRESULT WINAPI WsCreateHeap(
    _In_ SIZE_T maxSize,
    _In_ SIZE_T trimSize,
    _In_opt_ const WS_HEAP_PROPERTY* properties,
    _In_ ULONG propertyCount,
    _Outptr_ WS_HEAP** heap,
    _In_opt_ WS_ERROR* error);
```

WsCreateServiceProxy: crea el proxy para comunicarse con el servicio. Cuando se crea el proxy se ha de especificar en tipo de canal de datos (channelType) y el comportamiento de este (channelBinding)

```
HRESULT WINAPI WsCreateServiceProxy(
    _In_ const WS_CHANNEL_TYPE channelType,
    _In_ const WS_CHANNEL_BINDING channelBinding,
    _In_opt_ const WS_SECURITY_DESCRIPTION* securityDescription,
    _In_reads_opt_(propertyCount) const WS_PROXY_PROPERTY* properties,
    _In_ const ULONG propertyCount,
    _In_reads_opt_(channelPropertyCount) const WS_CHANNEL_PROPERTY*
channelProperties,
    _In_ const ULONG channelPropertyCount,
    _Outptr_ WS_SERVICE_PROXY** serviceProxy,
    _In_opt_ WS_ERROR* error);
```

WsOpenServiceProxy: abre el proxy apuntando a una dirección donde hay un servicio escuchando.

```
HRESULT WINAPI WsOpenServiceProxy(
    _In_ WS_SERVICE_PROXY* serviceProxy,
    _In_ const WS_ENDPOINT_ADDRESS* address,
    _In_opt_ const WS_ASYNC_CONTEXT* asyncContext,
    _In_opt_ WS_ERROR* error);
```

BasicHttpBinding_ICMPFSP_Service_SendSolucionG es el métido en el proxy encargado de llamar al método SendSolucionG en el servicio. A esta función se le ha de indicar el proxy abierto, el heap y la estructura SolucionGSer.

```
HRESULT WINAPI BasicHttpBinding_ICMPFSP_Service_SendSolucionG(
```

```
    _In_ WS_SERVICE_PROXY* _serviceProxy,
    _In_opt_ SolucionGSer* sg,
    _In_ WS_HEAP* _heap,
    _In_reads_opt_(_callPropertyCount) const WS_CALL_PROPERTY*
_callProperties,
    _In_ const ULONG _callPropertyCount,
    _In_opt_ const WS_ASYNC_CONTEXT* _asyncContext,
    _In_opt_ WS_ERROR* _error)
```

BasicHttpBinding_ICMPFSP_Service_DispatchSolucionG es el métido en el proxy encargado de llamar al método DispatchSolucionG en el servicio. A esta función se le ha de indicar el proxy abierto, el heap y la estructura SolucionGSer.

```
HRESULT WINAPI BasicHttpBinding_ICMPFSP_Service_DispatchSolucionG(
    _In_ WS_SERVICE_PROXY* _serviceProxy,
    _In_ int n,
    _Out_ unsigned int* DispatchSolucionGResultCount,
    _Outptr_opt_result_buffer_(*DispatchSolucionGResultCount) SolucionGSer***
DispatchSolucionGResult,
    _In_ WS_HEAP* _heap,
    _In_reads_opt_(_callPropertyCount) const WS_CALL_PROPERTY*
_callProperties,
    _In_ const ULONG _callPropertyCount,
    _In_opt_ const WS_ASYNC_CONTEXT* _asyncContext,
    _In_opt_ WS_ERROR* _error)
```

Comportamiento de las funciones implementadas:

A continuación se pueden ver los diagramas de secuencia que explican el comportamiento de las funciones ProxyIO.

Enviar

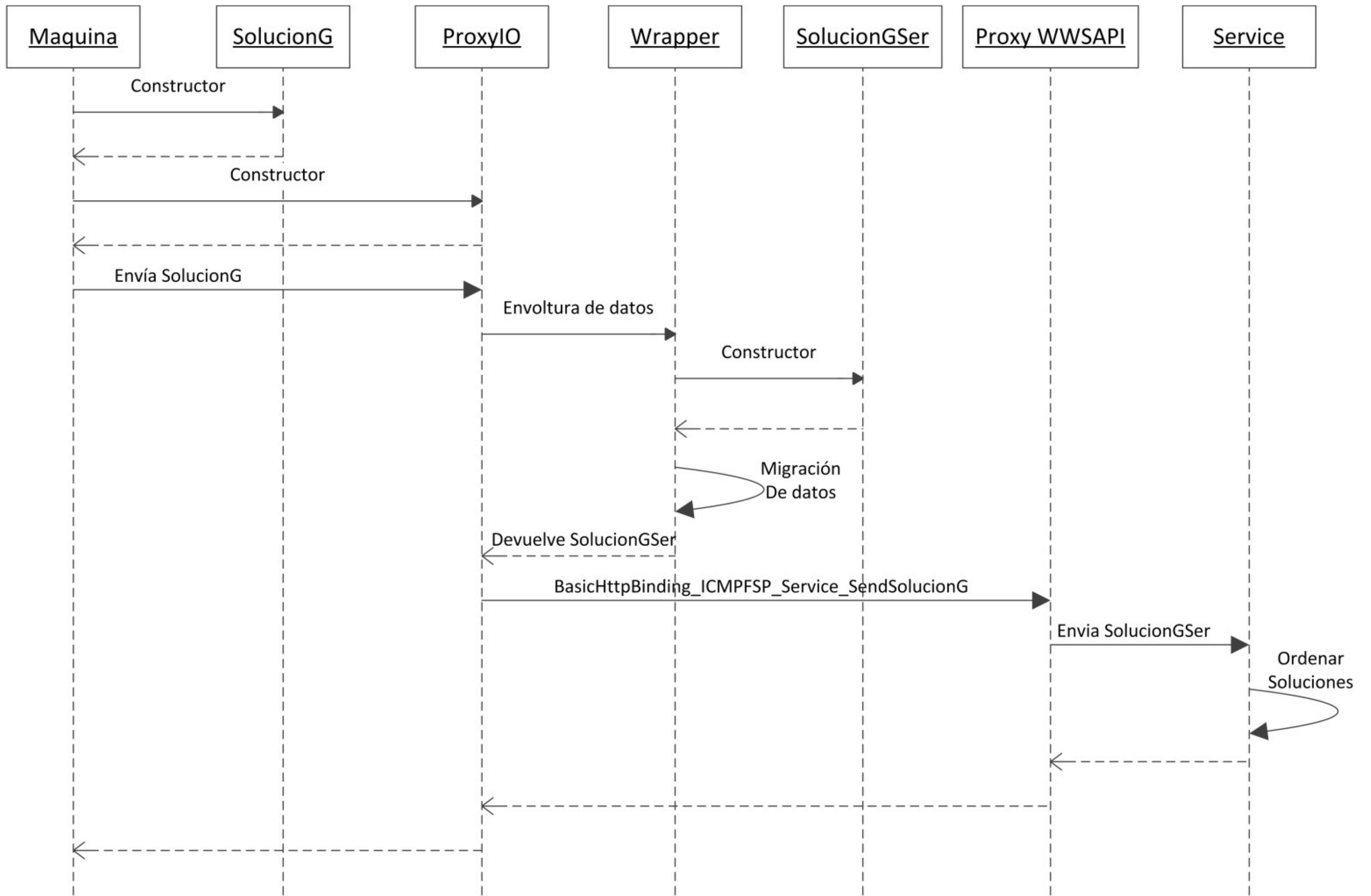
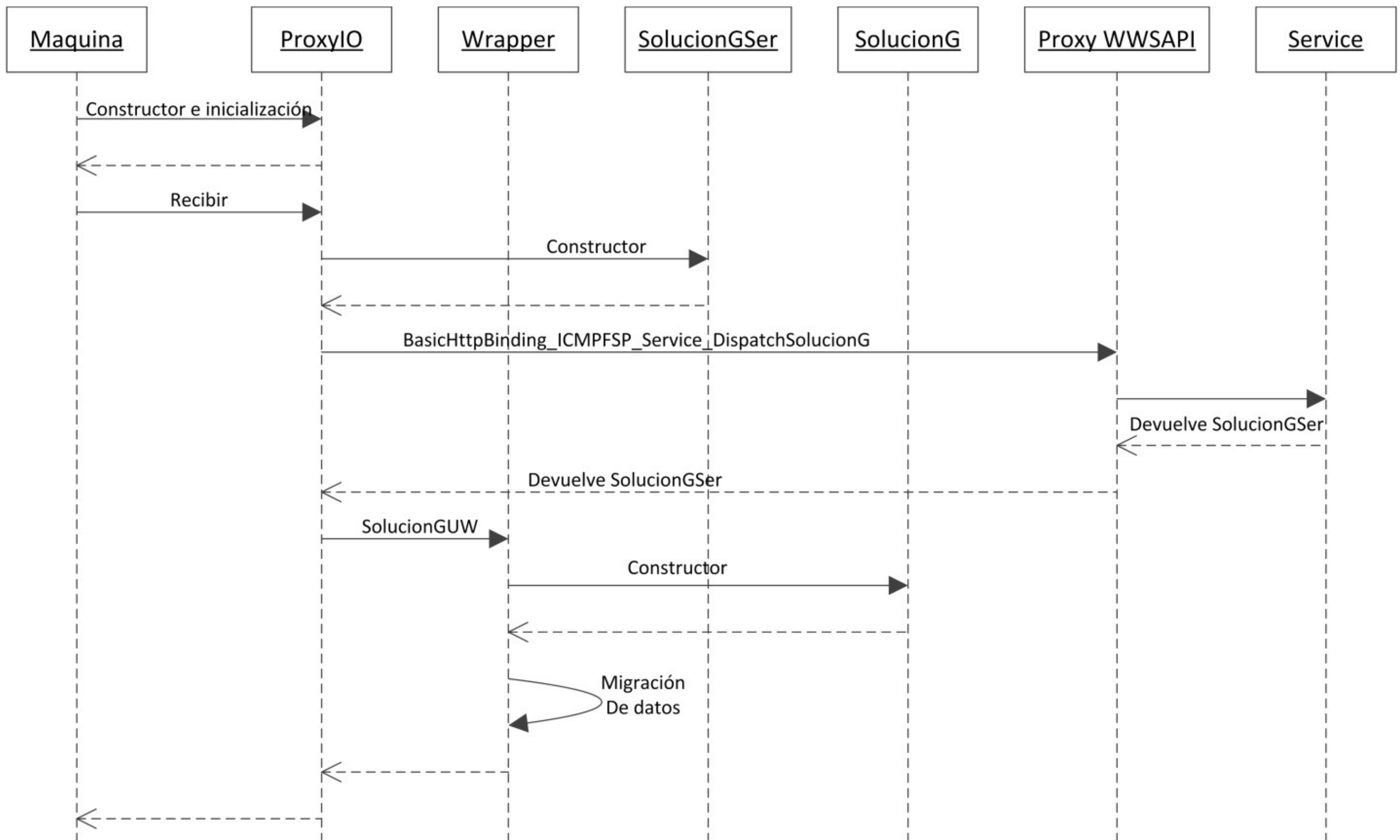


Figure 11 - enviar solución modelo maestro-esclavo

- Recibir

Figure 12 - recibir solución modelo maestro-esclavo



4.2.1.2 Islas

Este modelo no trabaja con proxis debido a la limitación de tener que configurar llamadas asíncronas desde el servicio al cliente obligándonos a trabajar con las funciones de la capa de canal de datos sobre un WS_CHANNEL_TYPE_DUPLEX_SESSION que ofrece comunicación dúplex de forma nativa. Lo primero que tendremos que hacer en el código es configurar los parámetros de seguridad de los mensajes tanto en el cliente como en el servicio ya que la configuración de los canales TCP por defecto en WWSAPI y WCF no es la misma, a continuación se crea y prepara el canal de datos por donde vamos a enviar y recibir mensajes con la información del endpoint en el servicio, después se crea el mensaje y por último se envía al servicio con los datos de la solución local y la descripción del mensaje que indica a qué función del contrato estamos llamando. Aparte de la función para enviar mensajes, es importante definir los métodos de suscripción para llevar un control de los destinatarios del

broadcasting, en este método simplemente se envía un mensaje con la descripción de la función suscripción del contrato y se crea una hebra por donde recibiremos los datos enviados por el servidor para implementar la funcionalidad asíncrona deseada.

Los mensajes a implementar en este modelo son:

Darse de alta: En esta función el cliente envía un mensaje al servicio con la información del canal de datos. El servicio usará la información del canal recibido para llamar la función callback y enviar los valores de vuelta. Además el cliente creará una hebra de ejecución para capturar los mensajes callback desde el servidor.

Darse de baja: El servidor simplemente da de baja la sesión con el cliente.

Enviar: El cliente envía un mensaje con su solución al servidor.

A continuación se puede consultar la definición formal de ProxyIO para el modelo de islas.

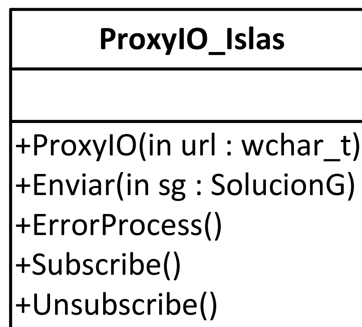


Figure 13 - Clase ProxyIO Islas

Funciones WWSAPI utilizadas en este modelo:

WsCreateChannel crea el canal de datos, se necesita especificar el tipo de canal de datos y el comportamiento de este.

```
HRESULT WINAPI WsCreateChannel(
    _In_ WS_CHANNEL_TYPE channelType,
    _In_ WS_CHANNEL_BINDING channelBinding,
    _In_reads_opt_(propertyCount) const WS_CHANNEL_PROPERTY* properties,
    _In_ ULONG propertyCount,
    _In_opt_ const WS_SECURITY_DESCRIPTION* securityDescription,
    _Outptr_ WS_CHANNEL** channel,
    _In_opt_ WS_ERROR* error);
```

WsOpenChannel especifica la dirección del servicio y abre el canal de datos. Necesita especificar el tipo de canal de datos y el comportamiento de este.

```
HRESULT WINAPI WsOpenChannel(
    _In_ WS_CHANNEL* channel,
    _In_ const WS_ENDPOINT_ADDRESS* endpointAddress,
    _In_opt_ const WS_ASYNC_CONTEXT* asyncContext,
    _In_opt_ WS_ERROR* error);
```

WsCreateMessageForChannel crea un mensaje vacío para un canal de datos específico.

```
HRESULT WINAPI WsCreateMessageForChannel(
    _In_ WS_CHANNEL* channel,
    _In_reads_opt_(propertyCount) const WS_MESSAGE_PROPERTY* properties,
    _In_ ULONG propertyCount,
    _Outptr_ WS_MESSAGE** message,
    _In_opt_ WS_ERROR* error);
```

WsCreateHeap crea el heap para gestionar la memoria para serializar datos.

```
HRESULT WINAPI WsCreateHeap(
    _In_ SIZE_T maxSize,
    _In_ SIZE_T trimSize,
    _In_opt_ const WS_HEAP_PROPERTY* properties,
    _In_ ULONG propertyCount,
    _Outptr_ WS_HEAP** heap,
    _In_opt_ WS_ERROR* error);
```

WsSendMessage envía un mensaje al servicio, cada mensaje ha de tener una descripción de mensaje que indica la función en el servicio al que va dirigido, los mensajes viajarán por un canal de datos previamente creado y contendrá los datos a enviar _SendSolucionG que no es más que un objeto que contiene una instancia de SendSolucionGSer.

```
HRESULT WINAPI WsSendMessage(
    _In_ WS_CHANNEL* channel,
    _In_ WS_MESSAGE* message,
    _In_ const WS_MESSAGE_DESCRIPTION* messageDescription,
    _In_ WS_WRITE_OPTION writeOption,
    _In_reads_bytes_opt_(bodyValueSize) const void* bodyValue,
    _In_ ULONG bodyValueSize,
    _In_opt_ const WS_ASYNC_CONTEXT* asyncContext,
    _In_opt_ WS_ERROR* error);
```

WsReceiveMessage se define para recibir callbacks desde el serivio, al igual que WsSendMessage, necesitamos especificar el canal de donde recibiremos los mensajes que en nuestro caso es TCP ya que soporta comunicaciones dúplex de forma nativa evitando crear un canal extra con su respectivo comportamiento y dirección para recibir los mensajes del servidor.

```
HRESULT WINAPI WsReceiveMessage(
    _In_ WS_CHANNEL* channel,
    _In_ WS_MESSAGE* message,
    _In_reads_(messageDescriptionCount) const WS_MESSAGE_DESCRIPTION**
messageDescriptions,
    _In_ ULONG messageDescriptionCount,
    _In_ WS_RECEIVE_OPTION receiveOption,
    _In_ WS_READ_OPTION readBodyOption,
    _In_opt_ WS_HEAP* heap,
    _Out_writes_bytes_(valueSize) void* value,
    _In_ ULONG valueSize,
    _Out_opt_ ULONG* index,
    _In_opt_ const WS_ASYNC_CONTEXT* asyncContext,
    _In_opt_ WS_ERROR* error);
```

WsResetMessage los mensajes se han de resetaer cada vez que son formateados y enviados para que puedan ser reutilizados.

```
HRESULT WINAPI WsResetMessage(
    _In_ WS_MESSAGE* message,
    _In_opt_ WS_ERROR* error);
```

Las descripciones de los mensajes pueden verse a continuación:

Cada vez que enviamos un mensaje hemos de indicar su descripción pare identificar la función en el servicio destino.

```
tempuri_org_wsdl.messages.ICMPFSP_Service_Subscribe_InputMessage

tempuri_org_wsdl.messages.ICMPFSP_Service_Unsubscribe_InputMessage

tempuri_org_wsdl.messages.ICMPFSP_Service_SendSolucionG_InputMessage

tempuri_org_wsdl.messages.ICMPFSP_Service_DispatchSolucionG_OutputCallbackMes
sage
```

Comportamiento de las funciones implementadas:

A continuación se pueden ver los diagramas de secuencia que explican el comportamiento de las funciones ProxyIO.

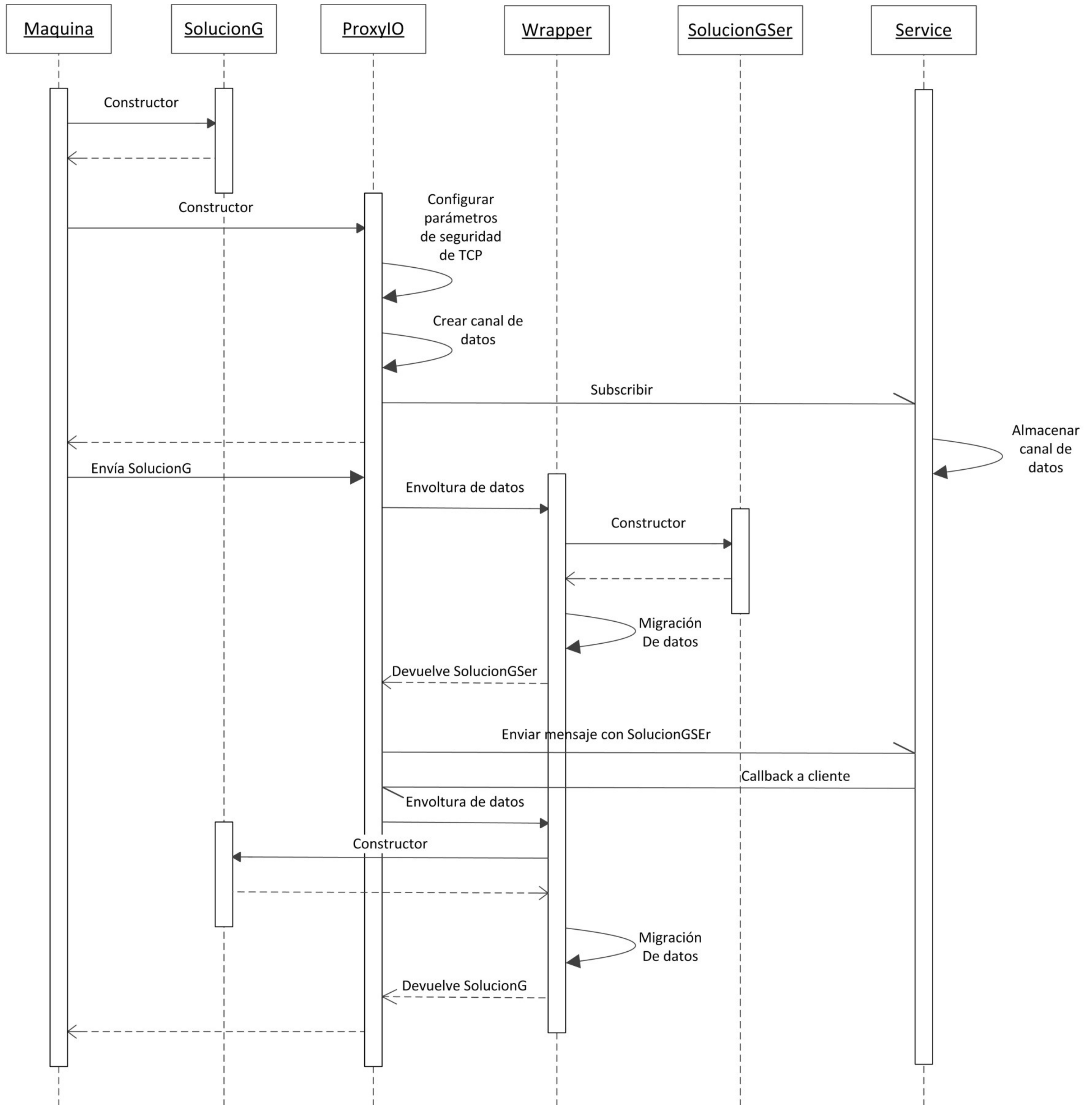


Figure 14 - Envío mensaje Islas

Nota: como ya apuntamos antes WWSAPI no da soporte para callbacks de forma nativa. El problema es que svcutil define los parámetros de las funciones callback como funciones de salida, una forma de solucionar esto es cambiar estos parámetros como de entrada y de esta forma wsutil creará las estructuras correctamente.

En conexiones TCP se han de abrir los puertos involucrados en los firewalls.

4.2.2 Librerías WWSAPI (ficheros wsdl y xsd)

4.2.2.1 Wsdl

Es un fichero en formato XML que describe servicios en una red de datos en forma de endpoints que trabajan con mensajes ya sea orientado a documento u orientado a procedimiento, las operaciones y los mensajes son descritos de forma abstracta que forman el punto de partida para crear proxis y stubs para que clientes y servicios puedan entenderse. En nuestro caso el fichero WSDL se crea automáticamente haciendo uso de Svcutil que se comentará más adelante.

4.2.2.2 Xsd

Estos ficheros definen las estructuras de los datos que viajan en los mensajes. Por ejemplo podemos encontrar los siguientes tipos de datos definidos:

Tipos básicos heredados de C: long, int, float, doublé, boolean, etc

Estructuras a serializar que serán enviadas como parámetros de las funciones en el servicio: SoluciónGSer, SMaquinaSer, JobSer, ArrayOfSMaquinaSer, ArrayOfJobSer or ArrayOfSolucionGSer.

4.2.3 Herramientas WWSAPI

4.2.3.1 SvcUtil

ScvUtil se utiliza para crear los ficheros que definen los metadatos (wsdl y xsd) del servicio, hay que tener en cuenta que el servicio ha de estar accesible desde un namespace al que SvcUtil tiene que acceder.

Como ya se comentó anteriormente WWSAPI no soporta callbacks, una de las razones de esto es porque la herramienta SvcUtil mapea los parámetros de estas funciones como parámetros de salida en lugar de parámetros de entrada. Si queremos crear las funciones en WWSAPI apropiadas para poder usar las funciones callback necesitamos cambiar los parámetros wsdl:output por wsdl:input en las funciones callback, en nuestro caso sería:

```
<wsdl:operation name="DispatchSolucionG">
```

<**wsdl:input** wsaw:Action="http://tempuri.org/ICMPFSP_Service/DispatchSolucionG" message="tns:ICMPFSP_Service_DispatchSolucionG_OutputCallbackMessage" />

</wsdl:operation>

Y

<wsdl:operation name="DispatchSolucionG">

<soap12:operation soapAction="http://tempuri.org/ICMPFSP_Service/DispatchSolucionG" style="document" />

<**wsdl:input**>

<soap12:body use="literal" />

</**wsdl:input**>

</wsdl:operation>

4.2.3.2 WsUtil

WsUtil es el compilador Windows para web services, creando la capa de servicios y serialización de los tipos de datos, esta utilidad procesa los ficheros WSDL y XSD generando las cabeceras y ficheros C, crea un proxy para el cliente y stub para el servicio si es necesario (es nuestro caso no necesitamos la parte de comunicaciones en la parte servicio puesto que esta es generada por WCF)

Hablar de las estructuras necesarias que forman el stub en el cliente.

Hablar de los cambios en el fichero wsdl para el callback

Ficheros creados por WsUtil

NOTA: los ficheros de ambos modelos (maestro-esclavo e islas) comparten mismo nombre y funcionalidad pero el código en ellos difiere debido a las distintas implementaciones de cada protocolo.

- schemas.microsoft.com.2003.10.Serialization.Arrays.xsd.h:
  Define las estructuras para serializar arrays de datos (struct ArrayOfint, truct ArrayOffloat, struct ArrayOfint y struct ArrayOffloat).

- schemas.microsoft.com.2003.10.Serialization.xsd.h:

  Define tipos básicos de C (boolean, datetime, decimal, doublé, etc.).

- WCF_CMPFSP.xsd.h:

Define las estructuras que serán serializadas y enviadas como parámetros de entrada al stub en el servicio (JobSer, ArrayOfSMaquinaSer, ArrayOfJobSer, ArrayOfSolucionGSer, SMaquinaSer, SolucionGSer).

- tempuri.org.xsd.h:

  Define los tipos de datos que esperan las funciones en el servicio. Por ejemplo _SendSolucionG es una estructura de datos que contiene SolucionGSer que es parámetro de entrada de la función SendSoluciónG en el servicio.

- tempuri.org.wsdl.h:

  Define las funciones publicadas por el contrato en el servicio y que podrán ser llamadas en nuestro código cliente como si fueran llamadas a funciones locales RPC. También define las descripciones de los mensajes que son mapeadas en el servicio con las funciones del contrato, por ejemplo en el caso del modelo de islas "tempuri_org_wsdl.messages.ICMPFSP_Service_Subscribe_InputMessage" indica que estamos enviando un mensaje a la función Subscribe en el servicio y por lo tanto los datos en el mensaje se han de mapear de acuerdo con los datos que espera la función en C#. Por última la definición de los proxis también puede encontrarse en este fichero.

Cada cabecera tendrá su correspondiente fichero fuente con el código implementado:

- schemas.microsoft.com.2003.10.Serialization.Arrays.xsd.c
- schemas.microsoft.com.2003.10.Serialization.xsd.c
- WCF_CMPFSP.xsd.c
- tempuri.org.xsd.c
- tempuri.org.wsdl.c

4.3 Persistencia de datos

Los datos han de ser persistentes en memoria durante la ejecución del algoritmo por lo que la declaración de las variables como estáticas será suficiente.

5 Implementación

5.1 Cliente

Máquinas

En este apartado sólo implementaremos el código en los clientes que se encargan de la comunicación, la parte del código encargada a la resolución de algoritmos ya está desarrollada.

5.1.1 Modelo maestro-esclavo - Proxy

El código del proxy ha sido desarrollado automáticamente usando las herramientas wsutil y svcutil de Visual Studio. Con svcutil sacaremos los ficheros .xsd y .wsdl que describen la metadata del servicio. Los proxis se crearán a continuación con wsutil y después añadiremos su código a nuestro cliente.

Ejecución y publicación del servicio

```
svcutil /t:metadata http://<servidor>:<puerto>/<nombre_del_contrato>

wsutil *.wsdl *.xsd
```

Incluimos los ficheros de nuestro proxy en el código cliente:

```
#include "schemas.microsoft.com.2003.10.Serialization.Arrays.xsd.h"
#include "schemas.microsoft.com.2003.10.Serialization.xsd.h"
#include "WCF_managed_app.xsd.h"
#include "tempuri.org.xsd.h"
#include "tempuri.org.wsdl.h"
```

Inicializamos estructuras en cliente para usar proxy:

```
hr = ERROR_SUCCESS;
error = NULL;
heap = NULL;
proxy = NULL;

WS_ENDPOINT_ADDRESS address = {};
address.url = url;

hr = WsCreateHeap(20000000, 20000000, NULL, 0, &heap, error);

WS_HTTP_BINDING_TEMPLATE templ = {};
```

NOTA: el heap es la memoria donde se almacenarán los datos durante el proceso de serialización y de-serialización, tendremos que darle un valor lo suficientemente grande como para poder deserializar los elementos que nos devuelve el servicio ya que de lo contrario no podremos espacio suficiente como para almacenar todas las SolucionesGSer que nos devuelve el servidor.

Creamos e inicializamos proxy

```
hr = BasicHttpBinding_ICMPFSP_Service_CreateServiceProxy(&templ, NULL, 0,
&proxy, error);

hr = WsOpenServiceProxy(proxy, &address, NULL, error);
```

Llamamos a las funciones que nos interesan

```
hr = BasicHttpBinding_ICMPFSP_Service_SendSolucionG(
    proxy, sgSer,
    heap, NULL, 0, NULL, error);



hr = BasicHttpBinding_ICMPFSP_Service_DispatchSolucionG(
    proxy, &result,
    heap, NULL, 0, NULL, error);
```

Envolturas

Las funciones para traducir clases del proxy al cliente se implementarán en una clase propia (Wrapper) para no contaminar el código de las Máquinas suministrado, es por eso que definiremos funciones estáticas que puedan ser llamadas desde el cliente.

Transforman nuestras estructuras de manera que las listas y vectores son migrados a arrays.

5.1.2 Modelo islas – Mensajes

Primero definimos los parámetros de seguridad en el cliente para poder establecer una comunicación con el servicio usando socket TCP y TcpBinding.

```
// Setup TCP security settings and bindings
WS_DEFAULT_WINDOWS_INTEGRATED_AUTH_CREDENTIAL defaultCred = {};
defaultCred.credential.credentialType =
WS_DEFAULT_WINDOWS_INTEGRATED_AUTH_CREDENTIAL_TYPE;

WS_SECURITY_BINDING_PROPERTY bindingProperties[1];
ULONG bindingPropertyCount = 0;

BOOL requireServerAuth = FALSE;
```

```
bindingProperties[bindingPropertyCount].id =
WS_SECURITY_BINDING_PROPERTY_REQUIRE_SERVER_AUTH;
bindingProperties[bindingPropertyCount].value = &requireServerAuth;
bindingProperties[bindingPropertyCount].valueSize =
sizeof(requireServerAuth);
bindingPropertyCount++;

WS_TCP_SSPI_TRANSPORT_SECURITY_BINDING sspiBinding = {};
sspiBinding.binding.bindingType =
WS_TCP_SSPI_TRANSPORT_SECURITY_BINDING_TYPE;
sspiBinding.clientCredential = &defaultCred.credential;
sspiBinding.binding.properties = bindingProperties;
sspiBinding.binding.propertyCount = bindingPropertyCount;

WS_SECURITY_BINDING* bindings[1] = {&sspiBinding.binding};

WS_SECURITY_DESCRIPTION securityDescription = {};
securityDescription.securityBindings = bindings;
securityDescription.securityBindingCount = WsCountOf(bindings);
```

El cliente crea su canal de datos:

```
hr = WsCreateChannel(
    WS_CHANNEL_TYPE_DUPLEX_SESSION,
    WS_TCP_CHANNEL_BINDING,
    NULL,
    0,
    &securityDescription,
    &channel,
    error);
```

Definimos la dirección del servidor que espera las peticiones y abrimos el canal de datos:

```
address.url.chars = L"net.tcp://localhost:9000/CMPFSP";
address.url.length = (ULONG)::wcslen(address.url.chars);
address.headers = NULL;
address.extensions = NULL;
address.identity = NULL;

hr = WsOpenChannel(
    channel,
    &address,
    NULL,
    error);
```

Creamos un mensaje para enviar datos al servicio:

```
hr = WsCreateMessageForChannel(
    channel,
    NULL,
    0,
    &message,
    error);
```

Subscribimos el cliente en el servidor y creamos el heap para recibir datos serializados:

```
Subscribe();

hr = WsCreateHeap(
    /*maxSize*/ 20000000,
    /*trimSize*/ 20000000,
    NULL,
    0,
    &heap,
    error);
```

El proceso de subscripción en el cliente está formado de la siguiente forma:

La primera parte simplemente envía un mensaje al servicio con la descripción adecuada:

```
hr = WsSendMessage(
    channel,
    message,
    &tempuri_org_wsdl.messages.ICMPFSP_Service_Subscribe_InputMessage,
    WS_WRITE_REQUIRED_VALUE,
    NULL,
    0,
    NULL,
    error);
```

Una segunda parte consiste en crear una hebra con un bucle infinito para recoger todas las peticiones de callback que le llegan desde el servidor.

Primero definimos las descripciones de los mensajes que recibiremos, en este caso sólo tendremos una descripción para la única función (`DispatchSolucionG`) de la interfaz callback.

Nota: la solución se almacenará en la variable sgVector definida como variable global. Cada vez que recibamos un mensaje llamaremos a la función `WsReceiveMessage` que nos devolverá los datos de la solución en la variable `result`. La variable `index` contiene el índice de la función llamada desde el servidor, en nuestro caso index sólo puede valer 0 puesto que sólo tenemos una función. Una vez `WsReceiveMessage` ha terminado su ejecución, pasaremos a a procesar el código de nuestra función callback que simplemente transformará los datos a `SolucionG` y los introducirá en `sgVector`.

```
const WS_MESSAGE_DESCRIPTION* messageDescriptions[] =
{
&tempuri_org_wsdl.messages.ICMPFSP_Service_DispatchSolucionG_OutputCallbackMe
ssage,
};
for (;;)
{
    void* result;

    ULONG index = 0;
    hr = WsReceiveMessage(
        threadInfo->channel,
        threadInfo->message,
        messageDescriptions,
```

```
        WsCountOf(messageDescriptions),
        WS_RECEIVE_OPTIONAL_MESSAGE,
        WS_READ_REQUIRED_POINTER,
        heap,
        &result,
        sizeof(result),
        &index,
        NULL,
        error);
    if (FAILED(hr) || WS_S_END == hr)
    {
        goto Exit;
    }
    if (index == 0)
    {
        sgVector.clear();
        for (size_t i = 0; i < ((_DispatchSolucionG*)result)-
>lSolucionGSerCount; i++)
        {

sgVector.push_back(*Wrappers::SolucionGUW(*((_DispatchSolucionG*)result)-
>lSolucionGSer[i]));
        }
    }
```

5.2 Servicio

La parte en el servidor se dividirá en definir los DataContract (tipo de datos intermedio a enviar por el canal de datos), interfaz en el lado servidor para el modelo de islas, interfaz en el lado servidor para el modelo maestro-esclavo, interfaz en el lado cliente para definir callbacks, implementación de las interfaces en el lado servidor. En las interfaces definiremos las funciones públicas que pueden ser usadas por otros nodos de la red. WCF nos obliga a definir explícitamente las interfaces con las funciones callback del cliente en el servidor para poder

A continuación se describen las funciones implementadas en el servicio:

5.2.1 Modelo maestro-esclavo:

SendSolucionG: el servicio recibe una solución y la almacena en una lista ordenada

```
public void SendSolucionG(SolucionGSer solucionGSer)
{
    bufferSG.Add(solucionGSer);
    bufferSG.Sort();
}
```

DispatchSolucionG: el servicio al cliente que lo solicita las “n” mejores soluciones recibidas por todos los clientes hasta el momento. Una solución se considera mejor que otra si el valor Coste es menor.

```
public SolucionGSer[] DispatchSolucionG(int n)
{
    if (n == 0 || n > bufferSG.Count()) return bufferSG.ToArray();
    else return bufferSG.Take(n).ToArray();
}
```

5.2.2 Modelo Islas:

Subscribe: da de alta un cliente almacenando su canal de datos para hacer llamadas callbacks

```
public void Subscribe()
{
    ICMPFSP_Service_Callback callback =
OperationContext.Current.GetCallbackChannel<ICMPFSP_Service_Callback>();
    if (!subscribers.Contains(callback))
        subscribers.Add(callback);
}
```

Unsubscribe: da de baja un cliente.

```
public void Unsubscribe()
{
    ICMPFSP_Service_Callback callback =
OperationContext.Current.GetCallbackChannel<ICMPFSP_Service_Callback>();
    if (subscribers.Contains(callback))
        subscribers.Remove(callback);
}
```

SendSolucionG: función del modelo islas enviando a todos los clientes registrados la solución recibida como parámetro.

```
public void SendSolucionG(SolucionGSer solucionGSer)
{
    bufferSG.Add(solucionGSer);
    bufferSG.Sort();

    foreach (ICMPFSP_Service_Callback callback in subscribers)
    {
        callback.DispatchSolucionG(bufferSG);
    }
}
```

5.2.3 DataContract

En WCF las estructuras de datos a enviar entre el servidor y los clientes se han de definir por medio de los DataContract, como se ha comentado anteriormente, no podemos enviar las estructuras de los clientes al servidor directamente puesto que los tipos de datos en WCF no son compatibles con los de C++. Los nuevos datos intermedios son básicamente una versión equivalente de los datos en C++ pero cambiando las listas y vectores por arrays. Los DataContract y los DataMember nos indican las clases las propiedades en C# que serán serializadas.

5.2.4 Hosting

En la configuración del Host definimos el comportamiento del servicio, el tipo de socket sobre el que trabaja, la dirección desde donde los clientes se pueden comunicar, el MEX (metadata exchange) desde donde svcutil sacará la descripción de los metadatos, y los parámetros para sacar las trazas de comunicación. En WCF se puede hacer de tres tipos distintos:

- IIS: hosting corre a cargo de Internet Information Services mediante un fichero de configuración web.config. este tipo de hosting se ha ignorado por simplicidad en el sistema,
- Hosting interno con fichero de configuración: El hosting se hace en la misma aplicación mediante un fichero de configuración con formato XML llamado app.config, el hosting del servidor para el modelo de islas está hecho de esta manera. Se puede destacar la configuración del NetTcpBinding que define el comportamiento del socket TCP.
- Hosting interno en el código: el hosting de la aplicación se hace programando el código. El hosting del servidor en el modelo maestro-esclavo está hecho de esta manera.

5.2.5 Persistencia

Persistencia en memoria, no necesitamos base de datos, por lo que definimos los datos en el servicio como static para que perduren durante toda la ejecución.

6 Resultados y conclusiones

6.1 Pruebas locales

Para la realización de las pruebas locales crearemos dos instancias de clientes que interactúen con el servicio al mismo tiempo. Desarrollaremos dos pruebas en total, una por cada modelo de comunicaciones desarrollado. Las pruebas se ejecutarán en Visual Basic que se usó para la implementación del algoritmo. Para asegurarnos que ambos clientes envían y reciben estructuras al mismo tiempo pondremos un breakpoint al principio de la ejecución y otro al final de manera que podremos comprobar los datos enviados por el servicio.

6.1.1 Pruebas modelo maestro-esclavo

Crearemos una instancia SolucionG y varias copias donde lo único que cambiaremos es el valor del parámetro Coste_Problema.

Una de las instancias del cliente enviará las soluciones con coste: 100, 600 y 15.

La otra instancia enviará soluciones con coste: 35, 30, 500, 200 y 130.

Todas estas soluciones se deberían almacenar en una lista ordenada en el servicio que al final enviará a los clientes.

Primero iniciamos el servicio ejecutando el código desde Visual Studio:

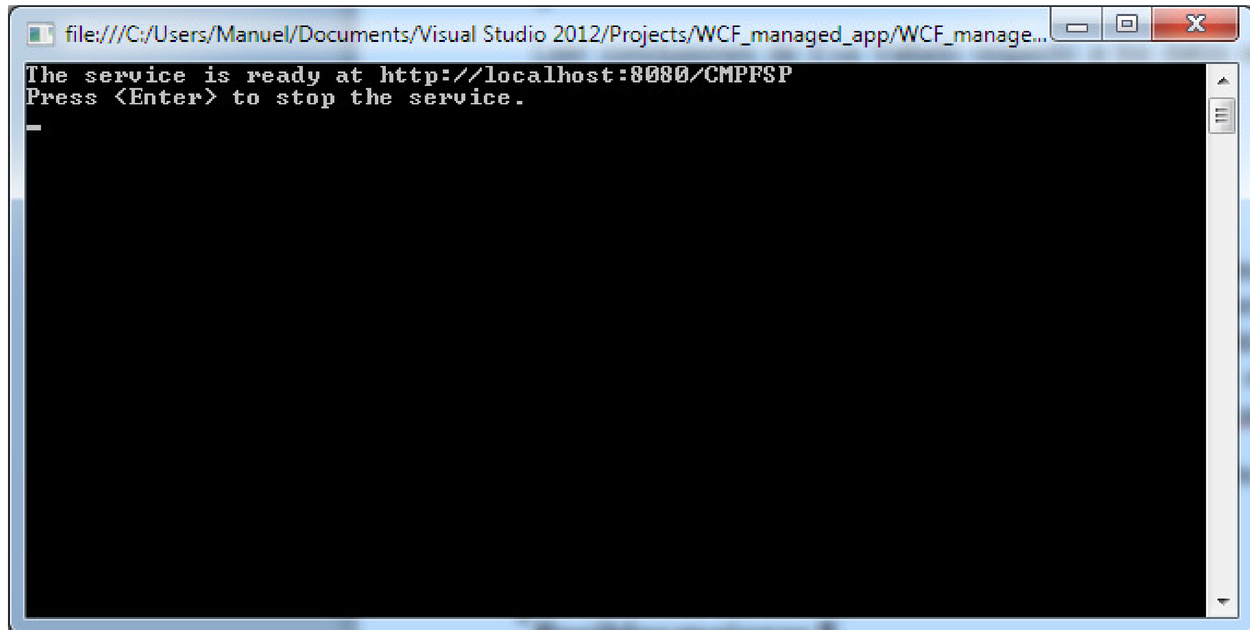


Figure 15 - ejecutar servicio

Una vez iniciado el servicio comprobamos que está preparado para aceptar peticiones de los clientes, para ello abrimos un navegador web y vamos a la dirección publicada http://localhost:8080/CMPFSP.

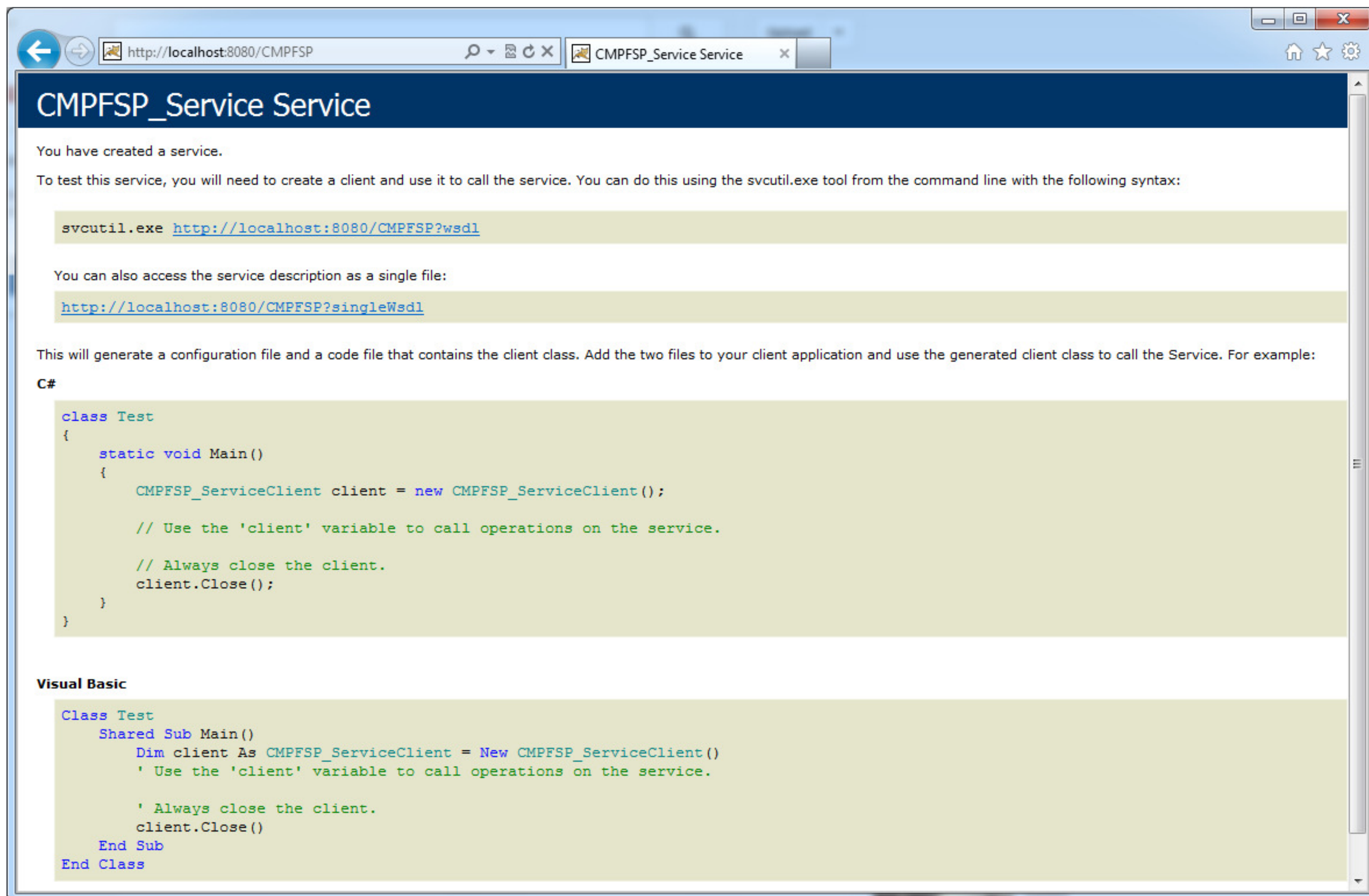


Figure 16 - comprobar servicio

Esta página web nos indica que el servicio está preparado, por lo que a continuación ejecutaremos los clientes al mismo tiempo.

Nota: en la dirección del WWSAPI proxy se ha de especificar la dirección del servicio que pusimos en el navegador web.

A continuación mostramos los valores de una de las instancias SolucionG antes de ser enviadas al servicio:

- sg {S_Problema={ size=2 } **Coste_Problema=100** r_asignacion=1 ...} SolucionG
- S_Problema { size=2 } std::list<SMaquina,std::allocator<SMaquina> >
- [0] {CsetB={ size=3 } CsetA={ size=3 } TJobs={ size=3 } ...} SMaquina
- CsetB { size=3 } std::list<Job *,std::allocator<Job *> >
- [0] 0x0017fd74 {ident=1 proc=1 alfa=1.10000002 ...} Job *

```
		ident	1	int
		proc	1	int
		alfa	1.10000002	float
		beta	1.10000002	float
		alfaN	-107374176.	float
		betaN	-107374176.	float
		Tiempo_total	-858993460	int
		final	0	int
		maquina	0	int
		EnA	0	int
		coste	0.00000000000000000	double
		costeA	-9.2559631349317831e+061	double
		costeB	-9.2559631349317831e+061	double
		ahorro	0.00000000000000000	double
		vecesB	0	int
		vecesA	0	int
		etiqueta	2	int
		etiqueta1	2	int
+		[1]	0x0017fd74 {ident=1 proc=1 alfa=1.10000002 ...}	Job *
+		[2]	0x0017fd74 {ident=1 proc=1 alfa=1.10000002 ...}	Job *
+		[Raw View]	0x002bdeb8 {...}	std::list<Job *,std::allocator<Job *> > *
+		CsetA	{ size=3 }	std::list<Job *,std::allocator<Job *> >
+		TJobs	{ size=3 }	std::vector<Job *,std::allocator<Job *> >
+		FinalJB	{ size=3 }	std::vector<int,std::allocator<int> >
+		FinalJA	{ size=3 }	std::vector<int,std::allocator<int> >
+		AlfasDcha	{ size=3 }	std::vector<int,std::allocator<int> >
+		BetasDcha	{ size=3 }	std::vector<int,std::allocator<int> >
+		RatioB	{ size=3 }	std::vector<float,std::allocator<float> >
+		RatioA	{ size=3 }	std::vector<float,std::allocator<float> >
		Cintermedio	0	int
		CTiempo_B	-858993460	int
		CTiempo_A	0	int
		CAlfas_B	0.000000000	float
		CBetas_A	0.000000000	float
		CCoste_Total	100	int
+		[1]	{CsetB={ size=3 } CsetA={ size=3 } TJobs={ size=3 } ...}	SMaquina
+		[Raw View]	0x0017fc44	{...}
	std::list<SMaquina,std::allocator<SMaquina> > *
		Coste_Problema	100	int
		r_asignacion	1	int
		q_insercion	1	int
		q_listas	1	int
		q_swapA	1	int
		q_swapB	1	int
```

Como puede verse Coste_Problema es 100. Esta estructura es transformada en SolucionGSer conservando todos susvalores y enviada al servicio mediante WWSAPI proxy.

Estos son los datos recibidos en el servicio:

```
-       solucionGSer  {WCF_CMPFSP.SolucionGSer}
    WCF_CMPFSP.SolucionGSer
        Coste_Problema  100   int
        q_insercion     1     int
        q_listas        1     int
        q_swapA         1     int
        q_swapB         1     int
        r_asignacion    1     int
-       S_Problema  {WCF_CMPFSP.SMaquinaSer[2]}
    WCF_CMPFSP.SMaquinaSer[]
-       [0]   {WCF_CMPFSP.SMaquinaSer}   WCF_CMPFSP.SMaquinaSer
-       AlfasDcha   {int[3]}  int[]
        [0]   1   int
        [1]   2   int
        [2]   3   int
+       BetasDcha   {int[3]}  int[]
        CAlfas_B     0.0   float
        CBetas_A     0.0   float
        CCoste_Total 0     int
        Cintermedio  0     int
+       CsetA  {WCF_CMPFSP.JobSer[3]}   WCF_CMPFSP.JobSer[]
+       CsetB  {WCF_CMPFSP.JobSer[3]}   WCF_CMPFSP.JobSer[]
        CTiempo_A    0     int
        CTiempo_B    -858993460   int
+       FinalJA      {int[3]}  int[]
+       FinalJB      {int[3]}  int[]
+       RatioA {float[3]}      float[]
+       RatioB {float[3]}      float[]
-       TJobs  {WCF_CMPFSP.JobSer[3]}   WCF_CMPFSP.JobSer[]
-       [0]    {WCF_CMPFSP.JobSer}      WCF_CMPFSP.JobSer
        ahorro 0.0   double
        alfa   1.1   float
        alfaN  -107374176.0 float
        beta   1.1   float
        betaN  -107374176.0 float
        coste  0.0   double
        costeA -9.2559631349317831E+61  double
        costeB -9.2559631349317831E+61  double
        EnA    0     int
        etiqueta     2    int
        etiqueta1    2    int
        final  0     int
        ident  1     int
        maquina      0    int
        proc   1     int
        Tiempo_total -858993460   int
        vecesA       0    int
        vecesB       0    int
+       [1]    {WCF_CMPFSP.JobSer}      WCF_CMPFSP.JobSer
+       [2]    {WCF_CMPFSP.JobSer}      WCF_CMPFSP.JobSer
+       [1]    {WCF_CMPFSP.SMaquinaSer}   WCF_CMPFSP.SMaquinaSer
```

Al final de la ejecución del cliente se solicitará todos los elementos recibidos por el servidor que se muestran a continuación.

```
-       sgVector        { size=6 }      std::vector<SolucionG,std::allocator<SolucionG> >
        [size]  6       int
        [capacity]      6       int
-       [0]     {S_Problema={   size=2  }  Coste_Problema=15  r_asignacion=1  ...}
    SolucionG
+       S_Problema      { size=2 }      std::list<SMaquina,std::allocator<SMaquina> >
        Coste_Problema  15      int
        r_asignacion    1       int
        q_insercion     1       int
        q_listas        1       int
        q_swapA         1       int
        q_swapB         1       int
-       [1]     {S_Problema={   size=2  }  Coste_Problema=30  r_asignacion=1  ...}
    SolucionG
+       S_Problema      { size=2 }      std::list<SMaquina,std::allocator<SMaquina> >
        Coste_Problema  30      int
        r_asignacion    1       int
        q_insercion     1       int
        q_listas        1       int
        q_swapA         1       int
        q_swapB         1       int
-       [2]     {S_Problema={   size=2  }  Coste_Problema=35  r_asignacion=1  ...}
    SolucionG
+       S_Problema      { size=2 }      std::list<SMaquina,std::allocator<SMaquina> >
        Coste_Problema  35      int
        r_asignacion    1       int
        q_insercion     1       int
        q_listas        1       int
        q_swapA         1       int
        q_swapB         1       int
-       [3]     {S_Problema={   size=2  }  Coste_Problema=100  r_asignacion=1  ...}
    SolucionG
+       S_Problema      { size=2 }      std::list<SMaquina,std::allocator<SMaquina> >
        Coste_Problema  100     int
        r_asignacion    1       int
        q_insercion     1       int
        q_listas        1       int
        q_swapA         1       int
        q_swapB         1       int
-       [4]     {S_Problema={   size=2  }  Coste_Problema=500  r_asignacion=1  ...}
    SolucionG
+       S_Problema      { size=2 }      std::list<SMaquina,std::allocator<SMaquina> >
        Coste_Problema  500     int
        r_asignacion    1       int
        q_insercion     1       int
        q_listas        1       int
```

```
                q_swapA         1       int
                q_swapB         1       int
-               [5]     {S_Problema={   size=2  }  Coste_Problema=600   r_asignacion=1   ...}
        SolucionG
+               S_Problema    { size=2 }       std::list<SMaquina,std::allocator<SMaquina> >
                Coste_Problema      600     int
                r_asignacion    1       int
                q_insercion     1       int
                q_listas        1       int
                q_swapA         1       int
                q_swapB         1       int
+               [Raw View]      0x003cf968                                          {...}
        std::vector<SolucionG,std::allocator<SolucionG> > *
```

Como puede verse el cliente ha recibido todos los elementos recibidos por el servidor, de forma ordenada.

6.1.2 Islas

Para el modelo de Islas haremos el mismo ejemplo que con el modelo maestro-esclavo. Crearemos dos clientes, cada uno creará distintas instancias de SoluciónG. Uno de los clientes creará las soluciones con Coste_problema 100, 500, 200, 130 y la otra 30, 600, 15.

En este método la recepción de las soluciones por parte de los clientes se hace de forma asíncrona mediante hebras creadas en ProxyIO, por lo que tendremos que añadir un retardo en el cliente para que no finalice su ejecución antes de recibir los datos del servicio.

El cliente almacenará las soluciones recibidas en un vector local.

En la siguiente captura se puede apreciar el momento antes de que el cliente finalice de ejecutar los elementos con los datos de sgVector que es donde el cliente almacena las soluciones recibidas del servidor.

WCF_callback_cp_bk (Debugging) - Microsoft Visual Studio (Administrator)

Quick Launch (Ctrl+Q)

EDIT VIEW PROJECT BUILD DEBUG TEAM SQL TOOLS TEST ANALYZE WINDOW HELP

Continue Debug Mixed Platforms

rocess: [3776] WCF_callback_Client_CPP_2 Suspend Thread: [3576] Main Thread Stack Frame: main

IService1.cs ProxyIO.cpp WCF_callback_Client_CPP.cpp WCF_callback_Client_CPP_2.cpp

(Global Scope) main()

```
    listSMaquina.push_back(sm);
    listSMaquina.push_back(sm);

    sg.S_Problema = listSMaquina;

    sg.Coste_Problema = 30;
    sg.q_insercion = 1;
    sg.q_listas = 1;
    sg.q_swapA = 1;
    sg.q_swapB = 1;
    sg.r_asignacion = 1;

    WCHAR url= WCHAR(L"net.tcp://localhost:9000/CMPFSP");
    ProxyIO proxy = ProxyIO(url);
    proxy.Enviar(sg);

    sg.Coste_Problema = 600;
    proxy.Enviar(sg);

    sg.Coste_Problema = 15;
    proxy.Enviar(sg);

    // waiste time so the client doesn't close the channel before service response
    int j = 0;
    for (int i = 0; i<10000000000000000000; i++)
    {
        j++;
    }

    j = 0;
    for (int i = 0; i<10000000000000000000; i++)
    {
        j++;
    }

    // Llegados a este punto deberíamos tener 7 elementos en sgVector;
    // 4 elementos de WCF_callback_Client_CPP y 3 de WCF_callback_Client_CPP_2
    std::wcout << L"-------------";

}
```

100 %

Autos Locals Threads Modules Watch 1 Parallel Watch 1 Call Stack Breakpoints Output

Name	Value	Type
sgVector	{ size=7 }	std::vector<SolucionG,std::allocator<Sol
[size]	7	int
[capacity]	9	int
[0]	{S_Problema={ size=2 } Coste_Problema=15 r_asignacion=1 ...}	SolucionG
S_Problema	{ size=2 }	std::list<SMaquina,std::allocator<SMaqu
[0]	{CsetB={ size=3 } CsetA={ size=3 } TJobs={ size=3 } ...}	SMaquina
CsetB	{ size=3 }	std::list<Job *,std::allocator<Job *> >
CsetA	{ size=3 }	std::list<Job *,std::allocator<Job *> >
TJobs	{ size=3 }	std::vector<Job *,std::allocator<Job *> >
FinalJB	{ size=3 }	std::vector<int,std::allocator<int> >
FinalJA	{ size=3 }	std::vector<int,std::allocator<int> >
AlfasDcha	{ size=3 }	std::vector<int,std::allocator<int> >
BetasDcha	{ size=3 }	std::vector<int,std::allocator<int> >
RatioB	{ size=3 }	std::vector<float,std::allocator<float> >
RatioA	{ size=3 }	std::vector<float,std::allocator<float> >
Cintermedio	0	int
CTiempo_B	-858993460	int
CTiempo_A	0	int
CAlfas_B	0.000000000	float
CBetas_A	0.000000000	float
CCoste_Total	0	int
[1]	{CsetB={ size=3 } CsetA={ size=3 } TJobs={ size=3 } ...}	SMaquina
[Raw View]	0x0050ec88 {...}	std::list<SMaquina,std::allocator<SMaqu
Coste_Problema	15	int
r_asignacion	1	int
q_insercion	1	int
q_listas	1	int
q_swapA	1	int
q_swapB	1	int
[1]	{S_Problema={ size=2 } Coste_Problema=30 r_asignacion=1 ...}	SolucionG
[2]	{S_Problema={ size=2 } Coste_Problema=100 r_asignacion=1 ...}	SolucionG
[3]	{S_Problema={ size=2 } Coste_Problema=130 r_asignacion=1 ...}	SolucionG
[4]	{S_Problema={ size=2 } Coste_Problema=200 r_asignacion=1 ...}	SolucionG
[5]	{S_Problema={ size=2 } Coste_Problema=500 r_asignacion=1 ...}	SolucionG
[6]	{S_Problema={ size=2 } Coste_Problema=600 r_asignacion=1 ...}	SolucionG
[Raw View]	0x00c8324c {WCF_callback_Client_CPP_2.exe!std::vector<SolucionG,std::a	std::vector<SolucionG,std::allocator<Sol

7 Conclusiones

WCF es una tecnología muy versátil y prometedora para implementar sistemas de comunicaciones entre algoritmos remotos, su integración con Visual Studio ayuda al desarrollo ágil de aplicaciones pudiéndose tener un sistema de comunicaciones a medida en pocas horas. Otro de los puntos fuertes de WCF es su documentación, la documentación oficial está muy bien desarrollada con multitud de ejemplos básicos que ayudan a su entendimiento y puesta en práctica, además también se pueden consultar los blogs de desarrollo de los empleados de Microsoft que explican y exponen ejemplos más complejos entrando con más detalle en las funcionalidades de WCF. Me ha sorprendido la facilidad con que los DataContract serializan los datos con estructuras en .Net, para entender cómo funciona esta funcionalidad, tuve primero que desarrollar los clientes en C# usando WCF y de esta manera aseguraba que el servicio estaba implementado de manera correcta, me fue relativamente fácil poder desarrollar el sistema de comunicaciones (cliente y servicio) en WCF puro, incluso implementado comunicaciones dúplex con los bindings DualHttpBinding.

Pero WCF también tiene sus limitaciones, uno de sus mayores problemas que encontré desarrollando este proyecto es que está fuertemente orientada a C#. Es difícil de integrar con C++ puesto que se necesita el uso de librerías WWSAPI para webservices perdiendo gran parte de la funcionalidad de WCF en el cliente como la serialización de colecciones en .Net o el soporte con comunicaciones dúplex. Incluso las herramientas que tiene Visual Studio para integrar los contratos de un servicio en un cliente, no están disponibles para aplicaciones en C++.

Uno de los mayores problemas vividos a la hora de desarrollar los algoritmos fue establecer la comunicación entre el servicio en C# y los clientes en C++, WWSAPI es una librería muy potente y versátil pero implica tiempo de desarrollo para implementar las estructuras intermedias que comuniquen el cliente y el servicio, además se requiere conocimientos de C y por lo tanto requiere un esfuerzo extra para implementar las estructuras en memoria adecuadamente.

Para concluir recomendaría WCF para desarrollar servicios en sistemas Windows, incluso aunque se tenga que desarrollar clientes en otros lenguajes, WWSAPI no es fácil de utilizar para comunicaciones complejas, pero una vez dominada esta tecnología te brinda muchas posibilidades siendo una alternativa bastante viable para proyectos de desarrollo a pequeña y gran escala.

8 Posibles mejoras.

Como alternativa de diseño, podríamos desarrollar la parte servicio en los clientes directamente, para el modelo de Islas, de esta forma evitaríamos de un sistema externo para hacer los callbacks y una llamada al servicio ofreciendo un entorno más próximo al mostrado en (VALLADA & RUIZ, 2007).

Otro cambio de diseño que se podría implementar sería el establecer la comunicación en el modelo de islas mediante socket HTTP, para ello necesitaríamos desarrollar un listener con su respectiva URL pública para recibir las llamadas del servicio, esto traería como ventaja el no tener que abrir puestos en los firewall en caso de comunicaciones en redes WAN y flexibilizar el modelo de islas ofreciendo otros sockets de comunicación.

9 Bibliografía

Windows Web Services API (Windows). (27 de November de 2012). Obtenido de Windows Dev Center - Desktop: http://msdn.microsoft.com/en-us/library/windows/desktop/dd430435(v=vs.85).aspx

ALVAREZ-VALDES, R., CRESPO, E., TAMARIT, J. M., & VILLA, F. (2011). Minimizing weighted earliness-tardiness on a single machine with a common due date using quadratic models. 14.

Dudar, M., & Dudar , M. (3 de March de 2009). *Demo of native code client to WCF services and native code web service using Windows Web Service API.* Obtenido de MSDN Blogs: http://blogs.msdn.com/b/nikolad/archive/2009/03/03/demo-of-wwsapi-from-mvp-summit.aspx

Eldos. (s.f.). Obtenido de http://www.eldos.com/msgconnect/

Fanjul Peyró, L. (2010). Nuevos algoritmos para el problema de secuenciación en máquinas paralelas no relacionadas y generalizaciones. 463.

Google. (s.f.). Obtenido de https://developers.google.com/protocol-buffers/

Microsoft. (s.f.). *#define VISUAL_STUDIO*. Obtenido de Visual Studio: http://msdn.microsoft.com/en-us/vstudio//

Microsoft. (October de 2006). *What You Need To Know About One-Way Calls, Callbacks, And Events*. Obtenido de MSDN Magazine: http://msdn.microsoft.com/en-us/magazine/cc163537.aspx#S6

Microsoft. (02 de 08 de 2012). *Developing Service-Oriented Applications - Windows Communication Foundation*. Obtenido de http://msdn.microsoft.com/en-us/library/dd456779.aspx

NADERI, B., & RUIZ, R. (s.f.). The distributed permutation flowshop scheduling problem. 32.

philpenn. (9 de July de 2009). *Implement Web Services with the Windows Web Services API.* Obtenido de MSDN: http://archive.msdn.microsoft.com/wwsapi

RIOS-SOLIS, Y., & SOURD, F. (2006). Exponential neighborhood search for a parallel machine scheduling problem. 16.

VALLADA, E., & RUIZ, R. (2007). Cooperative metaheuristics for the permutation flowshop scheduling problem. 12.

VILLA, F., ALVAREZ-VALDES, R., & M., T. J. (2011). A hybrid algorithm for minimizing earliness-tardiness penalties in parallel machines. 4.