# Comportamiento del middleware de comunicaciones Ice en entornos con y sin virtualización

Jorge Domínguez Poblete, Marisol García Valls, Christian Calva Urrego

Universidad Carlos III de Madrid
Av. de la universidad 30, 28911 Leganes, Spain
mvalls@it.uc3m.es

**Resumen** El middleware de comunicaciones tuvo su origen ya hace décadas, tiempos aquellos en los que el hardware existente (y la capacidad de las redes de comunicaciones) era sustancialmente diferente a las posibilidades de estas tecnologías en el momento presente. Concretamente, a medida que se ha introducido tecnologías de virtualización que permiten utilizar de forma más efectiva los servidores, se ha ido observando efectos a estudiar sobre los retardos de comunicación entre nodos. En este artículo se analiza el comportamiento de un middleware de comunicaciones concreto, Ice (Internet Communication Engine) que es una evolución de Corba. Se observan efectos interesantes y difíciles de explicar sobre todo para hardware con una capacidad de cómputo relativamente pequeña, en comparación a los grandes servidores típicos de centros de procesamiento en entornos de computación en la nube.

**Keywords:** middleware, prestaciones, software de virtualización, Internet Communications Engine (Ice)

## 1. Introducción

Los sistemas distribuidos actuales se ejecutan sobre nodos y redes cuyo hardware ha sufrido una evolución vertiginosa, sobre todo en la última década. La inundación del mercado con procesadores de gran capacidad de cómputo y de almacenamiento, la aparición de redes de comunicaciones de anchos de banda del orden de 100Gb, el avance de las tecnologías de virtualización con el consiguiente aumento en la consolidación de servidores [8], ha creado un entorno de ejecución radicalmente diferente al que encontró el middleware en sus orígenes.

En ciertas áreas de aplicación, como las relativas a los sistemas de tiempo real críticos, la utilización de middleware no está recomendada sobre todo en aquellas partes de nivel de criticidad mayor. Ello es debido a que tradicionalmente el middleware cuenta con una serie de librerías que utilizan técnicas de programación sofisticada más orientada a ofrecer un entorno de programación amigable, de gran productividad de programación, y comunicación fiable, eficiente y rápida en entornos de propósito general que a ofrecer garantías temporales estrictas.

 

Este es el caso para la gran mayoría de las tecnologías existentes como Corba [4], Java RMI [3], Ice [2], JMS [6], implementaciones de REST [7], AMQP [11], Storm [14], River [13], or JBoss [15], entre otros. Incluso el estándar OMG más popular en el momento como DDS [5] ofrece parámetros de calidad de servicio pero en ningún caso garantiza cotas máximas en los tiempos de comunicación. También Ada DSA [12] ofrece una extensión del lenguaje ada para sistemas distribuidos de tiempo real aunque está restringido al propio lenguaje.

Los sistemas distribuidos y de tiempo real y su evolución a los sistemas ciber-físicos requieren que los nuevos diseños de middleware de tiempo real tengan unas características específicas [18] que incluyan la verificación on-line de nuevas configuraciones [17]. Dentro de los sistemas distribuidos de tiempo real se han realizado varias contribuciones al diseño e implementación de middleware de tiempo real pero en su mayoría son aspectos que solucionan partes concretas de la predecibilidad en la comunicación. Por ejemplo, [9] es un middleware para sistemas distribuidos de tiempo real que requieren reconfiguración dinámica basado en gestión de calidad de servicio en los nodos [10] y algoritmos de reconfiguración [19]. [20] presenta un diseño inicial de middleware de tiempo real centrado en el ajuste de dos parámetros básicos y [21] presenta una arquitectura distribuida centralizada que soporta un número dinámico de clientes conectados a un servidor a través del manejo de su threadpool.

En este trabajo se analiza el comportamiento temporal de un middleware de comunicaciones de propósito general que surgió como una evolución de Corba. Este middleware es orientado a objetos, implementado en C++ y ofrece diferentes modelos de interacción, principalmente, basado en invocaciones remotas (RPC), con su consiguiente evolución a los objetos distribuidos, y en publicación-suscripción (P/S).

El artículo presenta la estructura que sigue. En la sección 2 se describe la arquitectura de Ice. En la sección 3 se analiza el comportamiento de Ice para un entorno sencillo (de ejecución directa sobre el hardware) y para un entorno virtualizado sobre KVM. La sección 4 presenta las conclusiones del trabajo y las líneas futuras y de continuación.

## 2. El middleware Ice

El diseño de Ice fue influenciado de forma clara por Corba pero con el objetivo de evitar incorporar los mismos errores de éste. Sin embargo, el API es bastante simple y muy fácil de usar. Su modelo de programación es potente ya que los objetos remotos pueden implementar diferentes interfaces remotas bajo una misma entidad de objeto. Permite utilizar comportamientos dinámicos enviando código de acceso al cliente y activando servidores bajo demanda para evitar un consumo de recursos cuando no es necesario.

### 2.1. Arquitectura y características

Ice es un framework de código abierto que soporta que máquinas o procesos cliente puedan comunicar entre sí estando implementados en múltiples (y dife-

 

rentes) lenguajes de programación como Java, C++, C#, PHP, Python, Ruby, y algunos más, y que puede ejecutar sobre varios sistemas operativos como Linux, Windows, OS X, Android, Solaris e iOS. Ice es un middleware de comunicaciones orientado a objectos que permite la creación de aplicaciones distribuidas cuya comunicación se basa en un protocolo de invocación a procedimiento remoto (RPC). El objetivo de los protocolos RPC, típicamente usados en comunicaciones del tipo cliente-servidor, es el de ejecutar funciones remotas (es decir, ubicadas en una máquina servidora, distinta de la máquina que invoca o cliente) de forma síncrona, devolviendo un resultado que puede ser nulo `void`.

Los middleware basados en RPC pueden utilizar diferentes lenguajes de programación en el servidor y en el cliente. Por ello, se establece un acuerdo explícito de las funciones remotas disponibles en el servidor para clientes a través de un *lenguaje de definición de interfaz* o IDL (*Interface Definition Language*) que en Ice se denomina *Slice*.

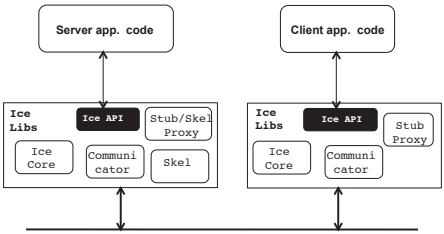


**Figura 1.** Vista esquemática de la arquitectura de Ice

La figura 1 muestra la arquitectura básica de Ice. Una aplicación o sistema distribuido con Ice contiene al menos una parte cliente y una parte servidora. Tanto cliente como servidor contienen código de nivel de aplicación y las librerías y run-time de Ice. Ice genera además código a partir de la definición de la interfaz Slice.

La parte indicada como **Ice Core** es el núcleo principal de la lógica del middleware que contiene el soporte del run-time para la comunicación remota tanto del lado cliente como del lado servidor. El *core* contiene, en su mayoría, los detalles de la red, concurrencia, ordenación de bytes y otros aspectos relacionados con la red que es aconsejable mantener alejados del código de aplicación.

Existe una parte del *core* que es dependiente de los tipos específicos declarados en Slice y utilizados en la comunicación entre cliente y servidor y otra parte que es independiente que se denomina **generic part of Ice core**. La parte genérica es accedida a través de el **Ice API**, que es idéntica tanto para clientes

 

como servidores y se encarga de tareas básicas de la administración de Ice como inicialización y finalización del run-time, entre otras tareas.

Para la gestión remota de la invocación, tanto en el lado del cliente como en el lado del servidor, aparecen unas entidades adicionales que reciben el nombre de **proxy**, concretamente son *stub* y *skeleton*. El código de los proxies se crea a partir de la definición de la interfaz IDL y, por lo tanto, son específicos de los objetos y datos que contiene la interfaz. *Stub* y *skeleton* son entidades que posibilitan (acercan) la comunicación con (y desde) los objetos remotos. Son, por tanto, representantes del objeto o código remoto que posibilitan que el cliente realice la traducción de las invocaciones a un formato neutro para su transmisión y el servidor pueda reconstruir estas invocaciones y las envíe hasta el nivel de aplicación como si se tratara de una invocación local.

El *stub* ofrece: (*i*) una interfaz para invocar los métodos de una interfaz de un objeto remoto; y (*ii*) código para *marshalling* y *unmarshalling*. Marshalling es el proceso de serializar o aplanar una estructura compleja de comunicación y convertirla en un formato que pueda ser transmitido por la red. El código de marshalling convierte una invocación a una secuencia de transmisión estándar que es independiente de las reglas de *endianness* y *padding* de la máquina local. El proceso de unmarshalling es lo contrario y deserializa los datos que llegan por la red, reconstruyéndolos a la representación local y tipos de datos apropiados para el lenguage de programación que se use. El *skeleton* es el código equivalente en el lado servidor, que ofrece una interfaz de llamada (up-call) para que el run-time de Ice transfiera el hilo de control al código de aplicación. También contiene el código de marshalling y unmarshalling de forma que el servidor puede recibir parámetros enviados por el cliente y devolver parámetros y excepciones al cliente.

### 2.2. Objetos específicos

El objeto *Communicator* o comunicador es el punto de entrada de todas las interacciones (es decir, invocaciones). Despacha las invocaciones a todas las facilidades y entidades de las librerías de Ice y controla el *thread pool*.

Un objeto *servant* o sirviente tiene la responsabilidad de gestionar y soportar las ejecuciones de las operaciones remotas del lado servidor ya que las invocaciones solicitadas se realizan en un lenguaje de programación específico.

Otra entidad importante de Ice es el objeto *adaptador*, que es específico del lado servidor. Su función es la siguiente [1]:

- Hace corresponder una invocación entrante al método específico de objetos de lenguage de programación. Por tanto, conoce qué sirvientes están cargados en la memoria y en qué objetos.
- Un adaptador tiene uno o varios puntos de entrada (*transport endpoint*). Si es el caso que un adaptador tenga varios transport endpoints, los sirvientes del adaptador podrán, a su vez, ser alcanzados vía múltiples transportes. Esto permite asociar dos *transport endpoints* a un mismo adaptador para ofrecer diferentes niveles de rendimiento y calidad de servicio.

 

- Es responsable de la creación de proxies que pueden pasarse a los clientes. El adaptador conoce el tipo, identidad y características del transporte de cada uno de sus objetos e include la información adecuada cuando el código de la parte servidora pide la creación de un proxy.

### 2.3. Seguridad

Una característica importante de Ice es que ofrece transmisión segura a través del protocolo SSL (*secure socket layer*), que es un estándar de facto para las comunicaciones en red seguras. Aunque en aplicaciones de propósito general el rendimiento experimentado no es excesivo, sí que debe ser considerado en entornos con restricciones de tiempo ya que combina una serie de técnicas criptográficas como: encriptación de clave pública, encriptación simétrica (de clave compartida), códigos de autenticación de mensajes y certificados digitales. Todo ello se traduce en que al establecer una conexión vía SSL se realiza un protocolo de *handshake* durante el cual, se validan los certificados digitales que identifican a los participantes en la comunicación y se intercambian las claves simétricas para encriptar el tráfico de la sesión. Debido a lo pesado de la utilización de encriptación de clave pública, ésta sólo se usa durante el espacio de handshake y cuando éste acaba, SSL utiliza códigos de autenticación de mensajes para asegurar la integridad de los datos. Esto permite que cliente y servidor comuniquen con garantías razonables de que sus mensajes están seguros.

## 3. Comportamiento temporal de Ice

En esta sección se describen las pruebas de rendimiento realizadas un sistema distribuido con Ice tanto con máquinas sin virtualizar como con máquinas virtualizadas.

En las pruebas *sin virtualizar* se utilizó un despliegue en dos máquinas Intel Celeron 3400 de doble núcleo a 2.6 GHz y 2GB y el sistema operativo Linux distribución Ubuntu 12.04.5 LTS de 32 bits y gcc v 4.6.3. Ice 3.4 y compilador gcc v 4.6.3.

Para el *entorno virtualizado* se utilizaron las mismas máquinas físicas y QUEMU Virtual CPU version 2.0.0 de 2 núcleos y 1GB RAM. Además se utilizó KVM en su versión asociada al kernel 3.19.0-49-generic #55 14.04.1-Ubuntu SMP. Se utilizó Ice v 3.5 y gcc version 4.8.4.

### 3.1. Código de ejemplo

En esta sección se muestra el código de un cliente que obtiene un proxy de un servidor y posteriormente realiza dos invocaciones remotas sobre dos objetos remotos. Este código tiene validez para todos los despliegues, tanto virtualizados como sin virtualizar.

 

```
ic=Ice::initialize(argc,argv);
Ice::ObjectPrx base= ic->stringToProxy("CService:default -h
localhost -p 10000");
CServicePrx remoteService=CServicePrx::checkedCast(base);
Ice::ObjectPrx base1 = ic->stringToProxy("CommService:default -
    -h
localhost -p 10001");
CommServicePrx remoteService1 = CommServicePrx::checkedCast(
    base1);
if (!remoteService)
throw "Invalid proxy";
remoteService -> server_op1();
remoteService1 -> server_op2();
```

La inicialización del run-time se realiza mediante un objeto *communicator* (`ic`) que permite acceder al núcleo mediante la función `stringToProxy` para obtener un proxy del objeto remoto con nombre público `CSService`. En este caso el objeto reside en la misma máquina local y en el puerto indicado (`10001`). Para invocar las operaciones del objeto remoto (`server_op1` y `server_op2`) se utiliza TCP, que se indica por defecto.

A continuación se muestra el código del servidor en el que se crean dos objetos remotos, cada uno en un adaptador diferente.

```
ic=Ice::initialize(argc,argv);
Ice::ObjectAdapterPtr adapter1 = ic->
 createObjectAdapterWithEndpoints("adapterA", "default -p
     10000");
Ice::ObjectPtr object1 = new CServiceI;
adapter1->add(object1, ic->stringToIdentity("CService"));
adapter1->activate();

Ice::ObjectAdapterPtr adapter2 = ic->
 createObjectAdapterWithEndpoints("adapterB", "default -p
     10001");
Ice::ObjectPtr object1 = new CommServiceI;
adapter1->add(object1, ic->stringToIdentity("CommService"));
adapter1->activate();

ic->waitForShutdown();
```

En la parte servidora se crea un objeto adaptador que permite que el servidor pueda publicar diferentes objetos servidores con distintas interfaces (`CServiceI` y `CommService`).

### 3.2. Resultados

En esta sección se muestran los resultados obtenidos para el sistema distribuido con Ice para los siguientes escenarios con el cliente y servidor en la:

- misma máquina física
- diferente máquina física
- misma máquina virtual
- diferente máquina virtual dentro de la misma máquina física
- diferente máquina virtual en diferente máquina física

En todos los casos se ha obtenido 10,000 medidas de invocaciones, mostrándose la media.

En la figura 2 se muestra el rendimiento en un escenario sin carga tanto en las máquinas físicas como en la red. Se puede derivar un comportamiento que corresponde, en general, al esperado, aunque no para todos los casos. A mayor tamaño de datos enviados, mayor es el tiempo que se tarda en procesar y transmitir la información. No existe una diferencia destacable entre los escenarios en los que la aplicación ejecuta en la misma máquina física o en la misma máquina virtual.

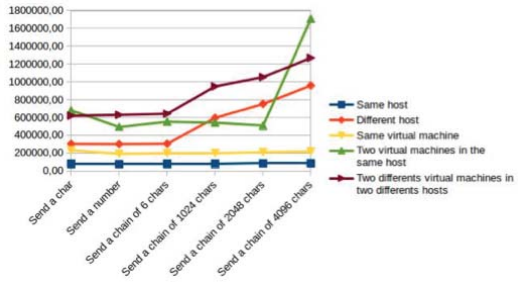


**Figura 2.** Rendimiento para un escenario sin carga

También se puede decir que el comportamiento es mejor de forma sostenida para el caso en que tenemos dos máquinas físicas sin virtualizar respecto al caso en el que se comunica dentro de la misma máquina física estando en cliente y servidor en máquinas virtuales diferentes.

En la figura 3 me muestra el rendimiento para un escenario con carga elevada, concretamente se ejecuta la parte servidora en una máquina con un 98 % de carga de forma sostenida. En este caso se observa que los escenarios de distinta máquina física y de dos máquinas virtuales en distintas máquinas físicas acusan esta saturación del sistema de forma más evidente.

Se observa también que los casos misma máquina física y misma máquina virtual tiene tiempos prácticamente iguales aunque existe un caso en el que el tiempo de comunicación supera al resto de casos previsiblemente peores.

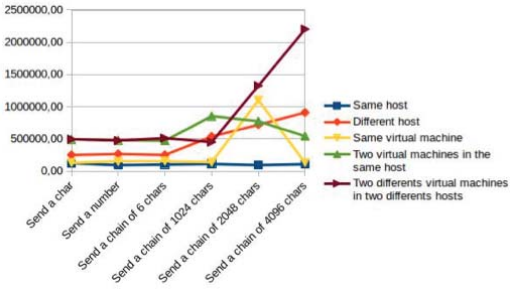


**Figura 3.** Rendimiento para un escenario con carga

En general, las máquinas utilizadas no poseen un hardware muy potente y, al no haberse utilizado ningún tipo de optimización para la virtualización, ello también influye en el comportamiento observado.

## 4. Conclusión

En este trabajo se ha presentado el middleware de comunicaciones Ice que ofrece un rendimiento excelente para aplicaciones distribuidas con requisitos de tiempo no estrictos. Se ha realizado una presentación de las características de este middleware y un análisis de su comportamiento temporal en entornos distribuidos tanto virtualizados como no virtualizados. Se observa que Ice presenta tiempos muy estables para la comunicación entre cliente y servidor en la misma máquina (con si sin virtualización). Sin embargo, en cuanto la distribución es real, el sistema acusa sobre todo la existencia de comunicación entre dos máquinas virtuales diferentes, siendo el rendimiento peor el observado en máquinas físicas diferentes que a su vez están virtualizadas. En presencia de carga elevada, el middleware no consigue mantener los tiempos de invocación en los valores normales y presenta mayor inestabilidad. Por tanto, se deduce que para cargas por debajo de 90 %, Ice presenta un comportamiento estable, ofreciendo tiempos de invocación sostenidos.

## Agradecimientos

This work has been partly funded by the project REM4VSS (TIN2011-28339) and M2C2 funded by the Spanish Ministry of Economy and Competitiveness.

 

## Referencias

1. ZeroC. Documentation for Ice 3.6. Client and Server Structure. https://doc.zeroc.com/display/Ice36/Client+and+Server+Structure (on-line) (2016)
2. ZeroC Inc.: The Internet Communications Engine. http://www.zeroc.com/ice.html (2003)
3. Sun Microsystems: Java$^{TM}$ Remote Method Invocation API http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/ (on-line) (2016)
4. Object Management Group: The Common Object Request Broker. Architecture and Specification, Version 3.3 (November 2012) http://www.omg.org/spec/CORBA/3.3
5. Object Management Group.: A Data Distribution Service for Real-time Systems Version 1.2. Real-Time Systems. (2007)
6. Deakin, N.: JSR 343: Java$^{TM}$ Message Service 2.0. Oracle. (2013)
7. Richardson, L., Ruby, S.: RESTful web service. O'Reilley Media. (2011)
8. García-Valls, M., Cucinotta, T., Lu, C.: Challenges in real-time virtualization and predictable cloud computing. Journal of Systems Architecture, 60(2): 726–740. (2014)
9. García-Valls, M.,Fernández-Villar, L., Rodríguez-López, I.: iLAND An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems. IEEE Transactions on Industrial Informatics, 9(1): 228–236. (2013)
10. García-Valls, M., Alonso, A., and de la Puente, J.A.: *A Dual-Band Priority Assignment Algorithm for QoS Resource Management*. Future Generation Computer Systems, 28(6): 902–912. (2012)
11. ISO/IEC Information Technology Task Force (ITTF). *OASIS AMQP1.0 – Advanced Message Queuing Protocol (AMQP), v1.0.* ISO/IEC 19464:2014. (2014)
12. Ada Core.: PolyORB. Ada Distributed Systems Annex (DSA). http://www.adacore.com/polyorb (on-line) (2016)
13. Apache Software Foundation. *Jini$^{TM}$ network technologies specifiation. Apache River v2.2.0.* (https://river.apache.org/doc/spec-index.html) (2013)
14. Apache Software Foundation. *Storm 0.10.0.* http://storm.apache.org (on-line) (2015)
15. JBoss. *JBoss Messaging*. http://docs.jboss.org (on-line) (2015)
16. Microsoft. *Disstributed Component Object Model (DCOM)*. (on-line) (2016)
17. Bersani, M. M., García-Valls, M.: The Cost of Formal Verification in Adaptive CPS. An Example of a Virtualized Server Node. 17$^{th}$ IEEE International Symposium on High Assurance Systems Engineering (HASE 2016), pp. 39-46. Orlando, FL. (2016)
18. García Valls, M., Baldoni, R.: Adaptive middleware design for CPS: Considerations on the OS, resource managers, and the network run-time. Proc. 14$^{th}$ Workshop on Adaptive and Reflective Middleware (ARM@Middleware). (2015)
19. García-Valls, M., Uriol-Resuela, P., Ibánez-Vázquez, F.: Low complexity reconfiguration for data-intensive service-oriented applications. Future Generation Computer Systems, vol.37. (2014)
20. García-Valls, M., Calva-Urrego, C., Alonso, A., de la Puente, J. A.: Adjusting middleware knobs to suit CPS domains. 31$^{st}$ ACM/SIGAPP Symposium on Applied Computing. Pisa, Italy. (2016)
21. García-Valls, M.: A proposal for cost-effective server usage in CPS in the presence of dynamic client requests. 19$^{th}$ IEEE Symposium on Real-time computing and distributed applications (ISORC). York, UK. (2016)

 