XXXVIII Jornadas de Automática
https://doi.org/10.17979/spudc.9788497497749.0942

Estudio e implementación de Middleware para aplicaciones de control distribuido

José-Enrique Simó-Ten, Jose-Luis Poza-Lujan, Juan-Luis Posadas-Yaguë, Francisco Blanes
Instituto de Automática e Informática Industrial (ai2)
Universitat Politècnica de València (UPV).
Camino de vera, s/n. 46022 Valencia (Spain)
jsimo@ai2.upv.es, jopolu@ai2.upv.es, jposadas@ai2.upv.es, pblanes@ai2.upv.es

Resumen

En este artículo se presenta un Middleware (CKMultipeer) desarrollado específicamente para la implementación de sistemas de control distribuido. El objetivo de este desarrollo es disponer de un sistema sencillo, con buenas propiedades sobre el que se puedan ensayar diferentes modelos de implementación tanto de las comunicaciones como de la ejecución de tareas. El Middleware desarrollado, además de permitir profundizar con implementaciones reales y diferentes estructuras de ejecución, se ha utilizado con éxito para la realización de diversas aplicaciones prácticas como es el caso de robots modulares que se han construido por agregación de componentes en red.

Palabras Clave: Control distribuido, Middleware, Publicación/Suscripción, Industria 4.0.

1 INTRODUCCIÓN

De una forma genérica, la tendencia que marca la cuarta revolución industrial, lo que se llama Industria 4.0, es el uso exhaustivo de las comunicaciones por red para la interconexión de componentes, desarrollados de forma independiente y con capacidad de adaptación. El objetivo final al que conduce esta tendencia es al desarrollo de sistemas industriales flexibles, que adapten su funcionamiento a las especificaciones de cada producto en particular y proporcionen y utilicen información de los niveles de planta y logísticos para optimizar la producción. Este objetivo final nos hace comprender que esta revolución está comenzando y queda mucho por hacer. En la actualidad, los fabricantes de automatismos ofrecen soluciones que conectan sistemas MES (*Manufacturing Execution System*) con ERP (*Enterprise Resource Planning*) consiguiendo simplificar la interacción entre el equipo humano y el sistema de producción mediante el uso exhaustivo de las tecnologías de la información. En esta evolución se encuentran paralelismos con los sistemas de "Internet de las cosas" [1] con los que comparte desafíos relacionados con el procesamiento masivo de la información para la búsqueda de patrones, sistemas de ejecución en la nube, coexistencia de diferentes niveles de criticidad, inestabilidad de las comunicaciones y cuestiones relacionadas con la ciberseguridad.

En este contexto, se pretende aprovechar las sinergias entre los sistemas distribuidos de fabricación y los avances en el campo de las tecnologías de la información.

Con la proliferación de sistemas de control distribuido derivada de la tendencia actual de desarrollo de la "Industria 4.0" es necesario un modelo de desarrollo que simplifique la implementación del conjunto de reguladores distribuidos que compone el control de la planta. En este sentido, este artículo presenta una solución Middleware (CKMultipeer) sencilla que permite ensayar las técnicas de control [2], de gestión de las comunicaciones, ejecución de tareas y monitorización de rendimiento más adecuadas para el desarrollo de sistemas en el marco de la Industria 4.0 e Internet de las Cosas.

2 TRABAJOS RELACIONADOS

Los sistemas distribuidos se implementan normalmente sobre algún tipo de middleware que extiende el marco de ejecución para proporcionar conexión entre dispositivos y mecanismo de intercambio de datos. Se pueden encontrar numerosas soluciones de comunicación tales como MQTT, *Java Message Service* (JMS) [3], *Internet Communication Engine* (ICE) [4], *Common Object Request Broker Architecture* (CORBA) [5], o *Distributed Data System (DDS)* [6]. Algunas de ellas como JMS proporcionan una arquitectura centrada en el mensaje mientras que otras como DDS se centran en los datos. Independientemente del paradigma que se utilice, todas las soluciones proporcionan

mecanismos confiables y flexibles para la comunicación asíncrona. JMS estable un API estándar común restringida al intercambio de mensajes en Java. DDS proporciona un sistema de Publicación/Suscripción independiente de la plataforma. Los entornos ICE y CORBA se basan en el uso de "mediadores de objetos" (Object Request Broker: ORB) que permiten el manejo de referencias a objetos distribudos. Las soluciones "Common ORB" basadas en el ORB de ACE, TAO [7] que se sustenta sobre el framework ACE (Adaptive Communication Environment) [8]. Para proporcionar al sistema tolerancia a fallos o mecanismos de adaptación es necesario que el middleware tenga mecanismos de monitorización y medida de indicadores de rendimiento. Algunos de las infraestructuras, como la citada TAO o DDS incorporan mecanismos de gestión de la calidad de servicio (QoS) [9]. Tal y como se introduce en [10] el diseño de un mecanismo de tiempo real de gestión de la calidad de servicio requiere la evaluación precisa del rendimiento del servicio y un marco específico para manejar las políticas de calidad de servicio [11].

3 REQUISITOS INICIALES

Como punto de partida, se ha considerado que el Middleware de comunicaciones tenga las siguientes características:

- El modelo de interacción entre elementos es uniforme y permite interacciones entre objetos que se encuentran en el mismo proceso, en diferentes procesos, pero en el mismo computador y entre objetos localizados en diferentes computadores.
- Modelo de comunicación "Publicación/ Suscripción"
- Comunicación basada en difusión asíncrona.
- Funciona sobre el protocolo basado en conexión TCP/IP.
- No tiene restricciones sobre el tamaño de la carga útil del mensaje
- Incorpora un servicio de descubrimiento implementado sobre el protocolo sin conexión TCP/UDP. Este protocolo detecta automáticamente los "pares" que forman parte del grupo de comunicación y los agrega para que reciban copia de todos los mensajes. En el caso en que el sistema tenga que funcionar sobre na red que no permita difusiones UDP, existirá un elemento "mediador" centralizado que gestionará el grupo, pero no intervendrá directamente en la comunicación.
- La comunicación entre los pares se realiza sin la intervención de componentes centralizados ni intermediarios. Todos los elementos tienen el mismo rol.
- Ofrece un reloj sincronizado entre todos lo pares.
- Los mensajes se envían con una marca de tiempo con resolución de microsegundos.
- Los mensajes se envían con una marca de prioridad de entrega.

4 DESCRIPCIÓN DE CKMultipper

Tal y como puede apreciarse en la Figura 1, el Middleware está estructurado en tres capas:

- Capa de aplicación
- Capa de sistema
- Núcleo de comunicación

La capa más baja constituye el núcleo de difusión de información. En esta capa se gestionan las conexiones entre los miembros del grupo y la difusión de mensajes al grupo. El formato de los mensajes es el que aparece en la Figura 2. Cuando se recibe un mensaje de red, se identifica el tipo de tráfico y se redirige hacia los puntos de procesamiento adecuados de la capa inmediatamente superior.

En la capa de sistema, se implementan servicios programando manejadores de mensajes que se asocian a determinados tipos de mensajes. De esta forma, se puede usar la misma infraestructura de núcleo de comunicaciones para implementar de forma totalmente separada diferentes servicios.

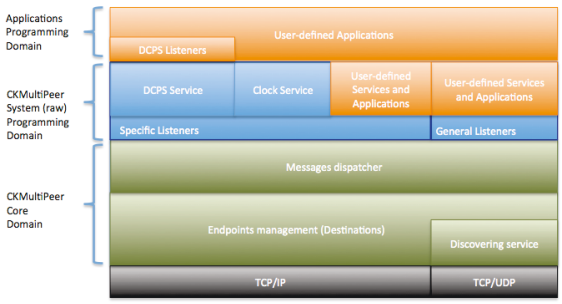

Figura 1: Estructura general de CKMultipeer.

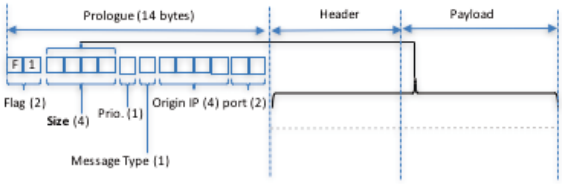

Figura 2: Formato del marco del mensaje.

4.1 NÚCLEO DE COMUNICACIÓN

Tal y como se ha comentado, el núcleo de comunicación se encarga de gestionar las conexiones entre los miembros del grupo, el protocolo de descubrimiento y la propagación de mensajes a través de las conexiones. Desde un punto de vista programático, esta funcionalidad la ofrece un objeto de la clase "**Multipeer**".

4.1.1 Conexiones entre los miembros del grupo

Cada uno de los miembros del grupo mantiene conexiones TCP/IP con el resto de los miembros formado un "mandala" de canales de comunicación bidireccionales. Para formar el entramado de conexiones, cada miembro acepta conexiones por un determinado puerto incorporando para ello un hilo de ejecución "servidor". Si al intentar crear el socket servidor en un puerto no lo consigue por estar ocupado, lo intentará con el siguiente puerto hasta que lo consigue (máximo 100 intentos). Una vez establecido el puerto por el que el par acepta conexiones, el par establece su identidad como un número de 64bits (GUID) calculado mediante la combinación de la dirección IP (32 bits sin signo) y el puerto en cuestión (16 bits) así: *((IP << 16) + puerto*).

El GUID de un par, además de identificarlo de forma unívoca en la red, se utiliza para establecer un orden entre los pares. Este orden sirve para crear de forma ordenada el entramado de conexiones: simplemente un par con GUID = $GUID_{actual}$ establecerá conexiones con aquellos pares detectados que tengan un GUID mayor que el suyo (más adelante se explicará el mecanismo de descubrimiento de pares) y dejará que los pares con GUID < $GUID_{actual}$ tomen la iniciativa de la conexión.
Las conexiones establecidas de esta manera se mantienen en el objeto "MultiPeer" mediante un conjunto de objetos "Destination" (en la forma de un objeto colección de la clase "PeerSet"). La clase "Destination" ofrece un método público "send()" para enviar mensajes y un método privado "receive()" para recibirlos.

4.1.2 Propagación de mensajes

Cada objeto "Destination" tiene un hilo de ejecución que se bloquea llamando al método "receive()" a la espera de recibir nuevos mensajes por la conexión. Cuando se recibe un mensaje, el hilo del objeto "Destination" invoca al método "OnMessage()" del objeto "MultiPeer" que encola el mensaje en una estructura de múltiples colas ("QueueMessage"). El mecanismo despachador de mensajes del objeto "MultiPeer" se encargará de procesar los mensajes de esta cola invocando las *callback* de los observadores que se hayan definido (Figura 3).

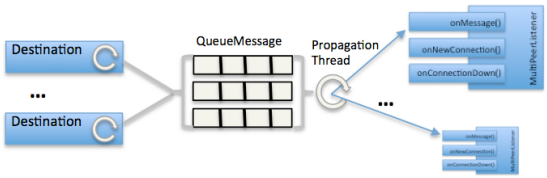

Figura 3: Gestión de los mensajes que se reciben desde las conexiones de red.

De esta forma se consigue simplificar el modelo de planificación de los hilos de ejecución. Por una parte, se ejecutan en el sistema los hilos de procesamiento de las comunicaciones, de alta prioridad, que simplemente mueven la información a las colas de entrada, su ejecución está dirigida por los eventos de comunicación. El hilo de propagación convierte la especificación de ejecución de los manejadores de eventos de comunicación en una ejecución secuencial que puede ser analizada y controlada.

4.1.3 Mecanismo de descubrimiento

Al iniciarse un objeto MultiPeer, establece un canal de difusión *multicast* escogiendo una dirección IP (de tipo "D" desde 224.0.0.0 hasta 239.255. 255.255) y un puerto. Esta combinación de IP y puerto de difusión es la "identidad del grupo". Cada miembro del grupo difunde su identidad enviando su GUID (IP y puerto por el que acepta conexiones como una cadena de caracteres con el formato "XX.XX.XX. XX:PPPP") por este canal de difusión y recibe la identidad de todos los miembros del grupo por este mismo canal. Cuando un miembro del grupo recibe por el canal la identidad de un nuevo miembro, difunde la suya propia para asegurar que notifica su identidad a los nuevos miembros del grupo. Cuando no hay nuevos miembros en el grupo, el canal de difusión permanece en silencio, no obstante, cada 5 segundos envía su identidad por el canal a modo de "latido de corazón" para comprobar que la infraestructura de red sigue siendo válida.
Cuando un miembro del grupo recibe la identidad de un nuevo miembro, aplica el mecanismo de conexión descrito anteriormente para formar el entramado de conexiones que constituirá la infraestructura de comunicación por la que circulan los mensajes de CKMultipeer.

4.2 CAPA DE SISTEMA

En la capa de sistema es donde se programan los servicios que utilizan el núcleo de comunicaciones. Cada servicio se implementa mediante la definición de uno o más tipos de mensaje y los correspondientes objetos de escucha (*Listeners*) que los procesa. El núcleo de comunicaciones se encarga de encaminar

los mensajes correctamente a los *Listeners* adecuados manteniendo totalmente separado el tráfico e los diferentes servicios.
Actualmente, el servicio más relevante que se ha programado sobre Multipeer es el modelo Publicación/Suscripción "DCPSMultipeer" que se detallará más adelante.
Además de la programación de nuevos servicios, un usuario-programador puede usar directamente la infraestructura Multipeer para difundir datos entre todos lo smiembros del grupo. Para ello se ha definido el tipo genérico de mensaje "PLAIN". El siguiente fragmento de código muestra un ejemplo de uso en C++:

```
#include "MultiPeerListener.h"
#include "Message.h"
#include "Destination.h"
#include "Clock.h"

class MPListenerTest: public MultiPeerListener {
public:
          MPListenerTest();
          virtual ~MPListenerTest();
          void onNewConnection(Destination* dest);
          void onConnectionDown(Destination* dest);
          void onMessage(Message* m);
};

MPListenerTest::MPListenerTest() {}

MPListenerTest::~MPListenerTest() {}

void MPListenerTest::onMessage(Message *msg) {
   long delay = Clock::currentMicros() - msg->getTimeStamp();
   printf("MPListenerTest: Mensaje recibido (delay = %ld): %s\n",
               delay,
               msg->getBufferPayload());
}

void MPListenerTest::onNewConnection(Destination* dest) {
   printf("MPListenerTest: Conexion recibida: %s\n",
              dest->getIPPort()->toString().c_str());
}

void MPListenerTest::onConnectionDown(Destination* dest) {
   printf("MPListenerTest: Conexion rota notificada: %s\n",
             dest->getIPPort()->toString().c_str());
}
//----
int main(void) {

   // Setup a MultiPeer Configuration Object
   MultiPeerConf mpCfg;
   mpCfg.setInterfaceName("eth0");
   //Build the communication artifacts
   MultiPeer mp(&mpCfg);

   MPListenerTest mpl;
   mp.addSpecificListener(MessageType::PLAIN,&mpl);

   IPPort *myself = mp.getLocalIPPort();
   unsigned char prio = NORMAL_MESSAGEPRIORITY;

   for(int i = 0 ;; i++) {
      string strMsg = string("Hola Multipeer desde ") +
                      mp.getLocalIPPort()->toString() +
                      string(" cuenta:") +
                      std::to_string(i);

      MessagePlain *msg = new MessagePlain(myself, prio,
                      (unsigned char *)strMsg.c_str(),
                      strMsg.size()+1);

      mp.send(msg);
      delete msg;
      usleep(100000);
   }
}
```

Como se puede deducir del código que se ha puesto como ejemplo, para programar en la capa de sistema es necesario tener ciertos conocimientos de la estructura interna del núcleo de comunicación y del uso de su API. No obstante, podemos observar que el ejemplo consiste en un programa principal que envía mensajes del tipo PLAIN a través de *Multipeer*, con una cadena de caracteres como carga útil. Al mismo tiempo establece un *Listener* (objeto de escucha) para este tipo de mensaje. El *Listener* recibe notificaciones cuando llega un mensaje (incluidos los que él mismo envía) y cuando se establece o rompe una conexión.

4.3 CAPA DE APLICACIÓN

La capa de aplicación es donde se desarrollan los módulos de control distribuido utilizando el modelo Publicación/Suscripción que ofrece la capa de sistema. Básicamente se define un módulo de control como un elemento que está interesado en recibir un conjunto de secuencias de datos (identificados mediante un nombre de registro o "Topic") y que produce un conjunto de resultados publicándolos en una serie de registros ("Topics").
Los nombres de los registros residen en un espacio de nombres ofrecido programáticamente mediante el objeto "TopicScope" que actúa como factoría de "Topics". Para poder recibir información de un Topic, es necesario definir un objeto de escucha (TopicListener) y suscribirlo al Topic en cuestión.
De una forma abstracta, se construye un sistema distribuido como el que se muestra en la Figura 4

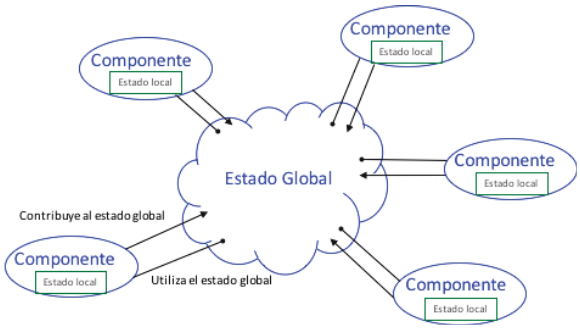

Figura 4: Estructura del sistema distribuido.

A modo de ejemplo, observe el siguiente código en C++:

```
#include "TopicListener.h"
#include "Topic.h"
#include "Clock.h"

class TopicListenerTest: public TopicListener {
public:
          TopicListenerTest();
          virtual ~TopicListenerTest();
          void onMessage(Topic* t, Message* m);
};

TopicListenerTest::TopicListenerTest() { }

TopicListenerTest::~TopicListenerTest() { }

void TopicListenerTest::onMessage(Topic* t, Message* m) {
   long delay = Clock::currentMicros() - m->getTimeStamp();
   printf("Topic(%s) %s (delay = %ld) Msg recibido(%ld): %s \n",
                     t->getName().c_str(),
                     m->getSourceIPPort()->toString().c_str(),
                     delay,
                     m->getPayloadSize(),
                     m->getBufferPayload());
}

//-----
int main(void) {

   MultiPeer mp;
   TopicScope scope(&mp);
   Topic *t = scope.lookup("DATA01");
```

```
    TopicListenerTest TopicListener;
    t->addListener(&TopicListener);

   char buffer[128];
   for(int i = 0;;i++) {
      sprintf(buffer,"12345 orden = %d",i);
      t->publish((unsigned char *)buffer,strlen(buffer)+1);
      usleep(50000);
   }
}
```

Funcionalmente el código es análogo al que se ha presentado anteriormente. El programa principal publica periódicamente información y se ha definido un objeto de escucha para esta información. En el código se puede observar cómo se define una infraestructura de comunicación (Multipeer) sobre la que se construye un espacio de nombres de datos (TopicScope) que actúa como factoría de "Topics".

Cuando se publica información en un "Topic", se notifica, mediante la ejecución del método "OnMessage" a cada uno de los objetos de escucha definidos en el sistema, independientemente de en qué ubicación se encuentren tanto la fuente como la escucha de la información.

4 IMPLEMENTACIÓN DE SOLUCIONES

El modelo descrito permite implementar de forma directa sistemas de control distribuido dirigidos por eventos ya que la infraestructura realiza una gestión transparente de la propagación de los eventos.

El modelo de propagación de eventos es bien conocido y analizable. En un determinado miembro del grupo, cuando se produce un evento, se pueden dar las siguientes posibilidades:

- **Evento local** (se produce en el mismo espacio de direccionamiento)
- **Evento externo** (se produce en espacio de direccionamiento diferente)
    - De un miembro en el mismo host
    - De un miembro localizado en un host remoto

Si el evento es local, el hilo que ejecuta la notificación (llamada a OnMessage) es el mismo que hizo la llamada "publish" sobre el Topic en cuestión. De esta manera se reduce drásticamente la latencia de notificación y se serializan las operaciones a realizar cuando se produce un evento. Con ayuda de la información de auditoría que proporciona el sistema, se es posible determinar la carga computacional que supone la atención a un evento.
Por otro lado, si el evento es remoto, está sujeto a la latencia de las comunicaciones entre procesos del mismo host o la de comunicaciones por red. En este caso el hilo de propagación es el encargado de serializar las operaciones y, análogamente al caso de eventos locales, es posible analizar la carga computacional con la información que proporciona el sistema.

Si se prefiere programar el sistema mediante procesos periódicos, CKMultipeer ofrece una serie de objetos de ayuda ("PayloadHolder") que permiten desacoplar la recepción de información de su uso. El siguiente código Java muestra un ejemplo de uso:

```
package dcpsTools;
import dcps.TopicsScope;

public class toolsTest {
   PayloadHolder dataHolder;
   int i = 0;
   public toolsTest() {
      TopicsScope ts = new TopicsScope();
      dataHolder =
            new PayloadHolder(ts.lookup("DATA01"));
      //Set the default value
      dataHolder.setPayload(new String("1425").getBytes());

       PeriodicTask t1 = new PeriodicTask(new Runnable()
{
            @Override
            public void run() {
            System.out.println("The DATA01 value is: " +
               new String(dataHolder.getPayload()));
            i++;

      }}, 5000000);

      t1.start();
   }

   ////////////////////////////////////
   public static void main(String[] args) {
      new toolsTest();
   }
}
```

El desarrollo de sistemas sobre CKMultipeer (y en general sobre cualquier middleware de comunicaciones) favorece una alta reutilización de código fuente y, lo que resulta más interesante, la reutilización de componentes ejecutables que se pueden adaptar simplemente configurando sus fuentes y destinos de información. En este sentido se han realizado experiencias de desarrollo de robots reconfigurables uniendo en una red interna del robot con conexión WiFi al exterior del robot, los siguientes elementos: Cámara, Xtion, IMU, Sensores IR, Control de motores, Unidad de navegación, y control de servos. Algunos de estos módulos se ejecutan sobre computadores convencionales pero la mayoría de ellos lo hacen sobre sistemas empotrados basados en Linux del tipo BeagleBone y RaspberryPi.

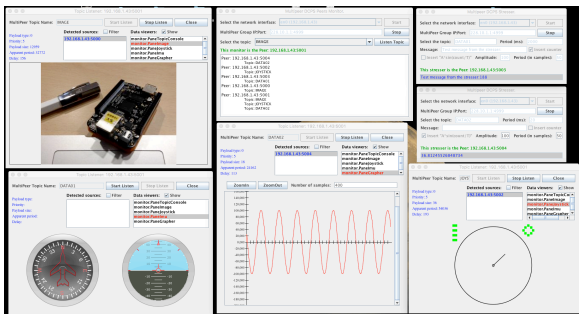

Figura 5: Herramientas de monitorización y visualización de información de CKMultipeer.

Actualmente existen implementaciones del sistema descrito (CKMultipeer) tanto en lenguaje C++ como Java, así como diferentes herramientas de monitorización, inyección y visualización de información. Ver Figura 5.

4 CONCLUSIONES Y TRABAJO FUTURO

En este artículo se ha presentado un middleware de comunicaciones orientado al desarrollo de sistemas de control distribuido. El desarrollo se ha realizado sin ninguna dependencia de código, lo que ha permitido estudiar las estructuras de ejecución y propagación de información más adecuadas para los propósitos del sistema.
Hemos comprobado que la versión actual del middleware permite desarrollar sistemas flexibles y reconfigurables en la línea de los objetivos establecidos en la tendencia de "Industria 4.0". No obstante, quedan por desarrollar importantes funcionalidades relacionadas con la gestión de la Calidad de Servicio y la inclusión de meta-información y su semántica asociada.
Las pautas de programación y los objetos de ayuda desarrollados permiten que se piense en herramientas de generación automática de código que no existen actualmente pero que se tiene previsto desarrollar en un futuro.

Agradecimientos

Este trabajo ha sido financiado parcialmente por el Gobierno de España (MICINN) a través del proyecto: M2C2: "Codiseño de sistemas de control con criticidad mixta basado en misiones" TIN2014-56158-C4-4-P.

Referencias

[1] Soma Bandyopadhyay, Munmun Sengupta, Souvik Maiti, and Subhajit Dutta. A survey of middleware for internet of things. In Recent Trends in Wireless and Mobile Networks, pages 288–296. Springer, 2011.

[2] Eduardo Munera, Jose-Luis Poza-Lujan, Juan-Luis Posadas-Yagüe, Jose Simo, J Francisco Blanes, and Pedro Albertos. Control kernel in smart factory environments: Smart resources integration. In Cyber Technology in Automation, Control, and Intelligent Systems (CYBER), 2015 IEEE International Conference on, pages 2002–2005. IEEE, 2015.

[3] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, and Kate Stout. Java message service. Sun Microsystems Inc., Santa Clara, CA, page 9, 2002.

[4] M Henning and M Spruiell. Distributed programming with ice. Technical report, ZeroC Inc., 2003.

[5] Ob ject Management Group (OMG). The Common Ob ject Request Broker (CORBA): Architecture and Specification. Technical report, Object Management Group, 1995.

[6] Object Management Group (OMG). Data Distribution Service for Real-time Sys- tems. Version 1 edition, 2007.

[7] DC Schmidt. TAO, The Ace Orb. Technical report, DOC group, Washington University, 2007.

[8] DC Schmidt. The adaptive communication environment. Technical report, DOC group, Washington University, 1994.

[9] Cristina Aurrecoechea, Andrew T Campbell, and Linda Hauw. A survey of qos architectures. Multimedia systems, 6(3):138–151, 1998.

[10] Woochul Kang, Sang Hyuk Son, and John A Stankovic. Design, implementation, and evaluation of a qos-aware real-time embedded database. Computers, IEEE Transactions on, 61(1):45–59, 2012.

[11] José L Poza, Juan L Posadas, and José E Simó. From the queue to the quality of service policy: A middleware implementation. In International Work-Conference on Artificial Neural Networks, pages 432–437. Springer, 2009.