Measuring performance of middleware technologies for medical systems: Ice vs AMQP

Paloma Rubio-Conde, Diego Villarán-Molina, Marisol García-Valls
Universidad Carlos III de Madrid, Spain
{parubioc, mvalls}@it.uc3m.es, diego.villaran@uc3m.es

ABSTRACT

After decades of design, development and usage of distributed application technologies, there are numerous communication middleware architectures and implementations in the market that have reached a considerable maturity level. A large number of them are open source initiatives that have shown efficiency and good performance in a broad range of domains, from banking to gaming. These are low cost solutions, easily programmable and of high interest to be explored in areas such as cyber-physical medical systems that have special requirements for safety, availability, communication latency, real-time operation, and fault tolerance. This paper analyzes the suitability of two open source communication middleware technologies, Ice (Internet Communication Engine) and AMQP (Advanced Message Queuing Protocol), as software elements suitable for developing audio transmission and reception systems for low cost medical applications. The paper simulates an audio application with both technologies, made of a server (nurse central) that receives and processes audio media from several clients (patients); communication can be triggered concurrent from multiple patients and in both directions. Stress tests with high load conditions are simulated in the experiments to show the behavior of both technologies mainly with respect to their stability and overhead.

Keywords

Cyber-physical medical systems; middleware

1. INTRODUCTION

Audio systems may become an important part of remote care medical systems, e.g., for remote monitoring of elderly patient health conditions. So, their impact can be at the level of reducing the costs incurred by the states in taking these patients to the hospital. These will be part of the application side in ICE medical architecture [17], possibly also in a virtualized context [4].

Audio applications for medical systems have time constraints for two main reasons: (a) inherited from their CPS context and their interaction with other physical monitoring and control elements, and (b) the inherent real-time nature of audio processing. In audio processing, a deadline miss in one sample creates noise that disturbs the perception and greatly annoys the final user.

Live audio-video systems are usually soft real-time systems where the violation of constraints results in degraded quality, but the system can continue to operate and also recover in the future using workload prediction and reconfiguration techniques [7, 8]. Digital signal processing (DSP) are part of the logic related to analyzing the input continuously along time; output frequency has to be stable independently of the processing delay [14]. The mean processing time per sample, including overhead, is no greater than the sampling period, which is the reciprocal of the sampling rate.

To support the development of distributed audio solutions, there are a number of powerful open-source middleware technologies that provide suitable performance to enable low cost solutions for remote patient monitoring. These can be applied in medical contexts where patients are not in critical conditions, e.g., elderly living on their own, but where their monitoring is needed and soft real-time interaction is required. Some proposals for architectures for medical systems such as ICE define the lowest level of communication to be DPWS[17]. Although this is suitable for health information systems to share documents and files through SOAP and XML, this is not a suitable approach for audio or video as the processing incurred by the parsing and additional message headers leads to high delays.

Therefore, not all middleware technologies are suitable as communication backbones in health care systems. For instance, in a centralised patient monitoring system that receives and displays the information about patients in their rooms/apartments, a bad decision on the selection of a middleware technology may yield to latencies of over tens of seconds.

This paper presents the characteristics of open source middleware technology and compares their efficiency in operation for acting as communication backbones. ZeroC Ice [25] and AMQP [12] are compared as they present two different architectural solutions that have a clear influence in their performance. A description of the design of the parallel audio processing from simultaneous clients is described for both, Ice and AMQP; finally, the results describing the execution time costs for both applications will be given as well as the conclusions for this work. Results should guide engineers to selecting a specific option based on its architecture and the specific off-line tests that should be executed.

The paper is structured as follows. Section II describes the context and related contributions in the area of distributed systems' technology for medical domains. Section III describes the base middleware technologies that are analysed and used in the paper. Section IV describes the design of the audio system and presents a general overview of their programming structure both in Ice and AMQP. Section V lays down the considerations to be made for allowing applications to have direct access to the execution platform resources, and how different middleware technologies may influence this aspect. Section VI presents the experimental results that compare the performance of both Ice and AMQP and their overhead in the audio system. Section VII concludes the work and relates conclusions to the obtained results.

2. BACKGROUND

The ICE [17] approach defines important elements such as the *protocol stack for medical device interoperability.* The lower layer of this stack is DPWS in charge of service discovery, interface description, messaging, event propagation, and secure information transmission. On top of this pure web service communication level, a streaming dual channel transmission based on MDPWS is provided; however the processing of the WS protocols is delay prone.

Below the DPWS layer, there is no further specification a part from the usage of HTTP/TCP or UDP. However, no specification of the basic communication middleware technologies is indicated. This is, precisely, the level at which the right technological choice leads to performance results of different orders of magnitude. So, this specific aspect deserves careful attention by the system engineers and an analysis of specific technologies must be done prior to actual design, development, and deployment. For a given standard (e.g. DDS[21]), different implementations from distinct vendors also lead to performance variations that may better adjust to given non-functional reuquirements.

There are a number of communication middleware backbones that are suitable for the ICE *messaging bus* that map to different interaction models: synchronous, asynchronous, remote invocations, publish/subscribe, or messaging. Examples are DDS[21], ZeroC Ice [25], Corba [20], Java RMI [23], River [1], JMS [3], stream processors [18], or iLAND [7] middleware for service oriented real-time applications, among others. To support timely operation in distributed domains, a number of contributions have appeared at different levels of the communication. Examples are: [2, 11] that enables higher dynaminity and real-time replacement of components on-line ; improvements to the number of clients supported by servers [9, 10]; and analysis of specific publish-subscribe communications over specific virtualized nodes [6].

In the case of the medical architecture defined by ICE, using DPWS bounds applications to SOAP protocols with messages in XML. This results in heavy communication latencies and parsing/unparsing times that may not be suitable to all domains, specially for those with timing constraints.

Studing alternatives to a pure DPWS backbone is needed as it yields to timely interaction between remote nodes/devices that is more suitable for the inherent temporal requirements of cyber-physical systems. Timeliness is a critical issue for some subsystems of a larger CPS that must be considered. Delays due to message parsing or heavy XML message transmission easily propagate to the control loop, which can destabilize the control over the physical system. Specific alternative solutions for efficient transmissions must be considered for the health domain based on the specific temporal requirements of applications.

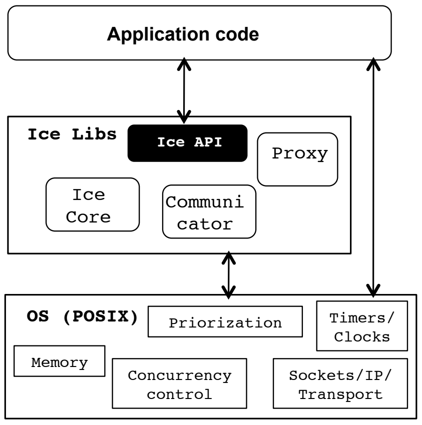

Figure 1: Ice architecture

3. COMMUNICATION MIDDLEWARE

3.1 Internet Communication Engine: Ice

The Internet Communications Engine (Ice) [25] (shown in figure 1) should not be confused with the medical systems architecture ICE [17]. Ice middleware is an object-oriented middleware platform that can be used to build distributed client-server or publish-subscribe applications in a simple way. It supports heterogeneous systems interconnection from the point of view of their hardware, operating system, and programming language. One important aspect is that it provides secure transmissions, and efficient network bandwidth utilization, memory usage and low CPU overhead due to its own operation internals and optimizations of its communication protocol messaging and transmissions.

Ice objects are the essential remote entities that can be instantiated in either a server or multiple servers. Each Ice object has one or more interfaces, which are collections of named operations that are supported by an object.

Ice design, presented in figure 1, was clearly influenced by Corba; however, the API has been maintained quite simple and it is extremely easy to use, inspired in Corba, but with the goal of avoiding the experienced mistakes of the latter. Its programming model is quite powerful as remote objets may implement different interfaces through a single object identify. It supports some level of dynamic behavior by sending proxies to clients, and activating servers on demand which favors performance when not needed.

The *Communicator* entity is the entrance point to all interactions. It dispatches requests to all facilities of Ice libraries. It controls the client-side and server-side thread pools. The *Adaptor* is a server-side entity that maps the requests to the server interface. The *Proxy* is an entity that is instantiated in the client that represents the remote object; it supports the remote invocations from clients as if they were local calls. The *Skeleton* is equivalent to the proxy but on the server side for translating incoming requests. The *Servant* is in charge of supporting the server-side operations to solve the requested invocations in a specific programming language.

Distributed applications can be programmed in different languages; one end can use C++, so that the resulting execution environment will be direct access, and it is possible to use all the system calls of the operating system; the other end can be programmed in C#; in such a case, it is needed to integrate an additional gateway run-time virtual machine.

3.2 Advanced Message Queuing Protocol

AMQP was originated in the financial services industry. It has general applicability to a broad range of application domains, including medical systems.

AMQP is an open standard application layer protocol and it is intended for message-oriented middleware. Main AMQP characteristics are message orientation, queuing, routing (publish-subscribe and point-to-point), security [8] and reliability.

The protocol defines the behaviour of the server that produces the messages (messaging provider) and the messaging client to the extent that implementations from different vendors are interoperable; in this way, it is similar to other protocols such as Http or Ftp, among others.

Previous middleware standardizations attempts were made at API level (i.e. JMS, Java Message Service), but they did not get a real interoperability [9] between multiple implementations. As opposed to JMS that only defines an API, AMQP is a wire-level protocol which is a description of the data format that are sent across the network as a byte stream. Consequently, any tool that can create and interpret messages according to this data format can interoperate with any other tool that complies to the protocol, regardless of the implementation language.

The main entities that are defined, from an interconnection point of view, are:

- *Message broker* is a server to which clients connect.
- *User* is an entity that, by submitting correct credentials (i.e. a password) is allowed to connect to a broker.
- *Connection* represents a transport protocol and connection linked to a user.
- *Channel* is a logical connection, having a state for each individual connection. Those clients who make concurrent operations through a single connection must maintain a different channel for each of these. Customers who use a thread-based model for concurrency can, for example, encapsulate the channel statement in a local variable for each thread.

The AMQP model is based on three entities: *exchange* (that receive messages from publishers and route them to message queues based on arbitrary criteria as message properties or content); *message queue* (entities receiving messages and storing them until they can be safely consumed by clients); and *binding* that are rules defining the relationship between a message queue and an exchange and provide the message routing criteria.

AMQP wire protocol definition allows for all common messaging behaviors. It does not define a wire-level distinction between clients and brokers, the protocol is symetric. However, different implementations may have different capabilities.

AMQP is rather oriented to providing a simple message bus for integration of systems. The variety of realizations yields to both mentioned run-time architectures: direct (over C++ and POSIX compliant OS) and gateway (e.g. Java). For example, RabittMQ implementation may use either a C# Mono environment or a C++/POSIX environment.

3.3 Differences between Ice and AMQP

The specific implementation of a middleware technology or specification has a determining influence on the overall performance. The specific used programming language must be also carefully considered. In this respect, by using Java interfaces, a virtual machine is added that relatively isolates the application execution. Applications that require to access the platform resources, e.g., to control elapsed time, etc., will then use run-times that directly compile to the platform such as C++; moreover, a POSIX interface enables finer control by using the kernel specific functions to access the platform resources.

4. AUDIO SYSTEM DESIGN

4.1 System overview

The audio system has been design and implemented for both paradigms: remote objects (Ice) and messaging (AMQP) in a homogeneous structure. Both patient side and server side can trigger the interaction protocol for audio transmission. Both middleware technologies are open source code implementations, easily programmable. Figure 2 shows the overall structure.

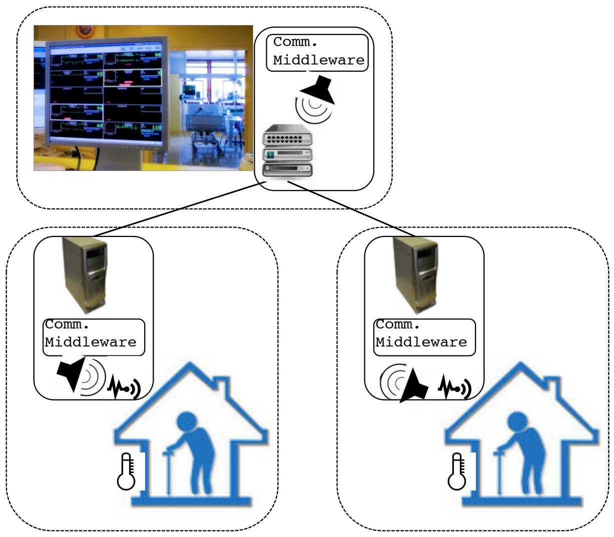

Figure 2: Overview of the application in a medical scenario

The goal of the system is to support multiple patients concurrently requesting service to transmit audio that has to be received at the server side. The server side processing can be done in parallel taking advantage of the physical hardware processing cores. Also, the server can prioritize the processing of specific patients due to different conditions: health characteristics, contracted service, or other considerations.

Priorization is then done in the patient request and in the server attention to the request of given patients.

4.2 Ice development

The system supports multiple patient communications with the server. Communication is based on sending packet audio frames of size 1024 bytes, where each byte is an audio sample. Therefore, each audio frame has 1024 samples that will be processed one by one at the server once it has received the entire frame. Once processed, the server sends an acknowledgement to the patient side. The design of the Ice application shown in figure 3.

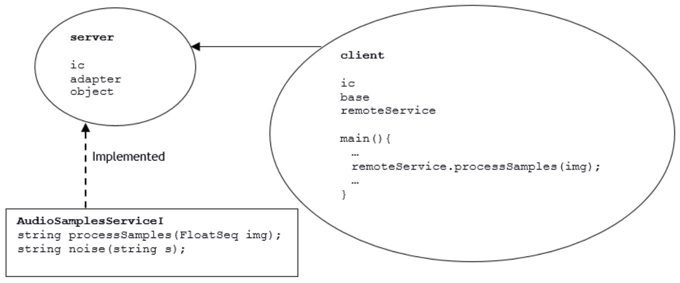

Figure 3: Software design for audio system based on object oriented middleware

The remote interface at the server side is independent of the specific programming language. The server side contains the interface (`AudioSamplesService.ice`). This interface has two functions that map to the two key functions of the server: processing samples (`processSamples`) and generating noise (`noise`), being the latter to introduce high load at the server side in order to test the stability of the communication and processing of the protocol stack. This interface is shown below.

```
module UC3M{
sequence<float> FloatSeq;
        interface AudioSamplesService{
                string processSamples(
                    FloatSeq img);
                string noise(string s);
        };
};
```

The patient side (`client.cpp` shown below) provides patient interfacing to the distributed application. It shows how to easily locate the server application side by using the Ice object `ObjectPrx` with the specification of the remote server IP address.

```
int main(int argc, char* argv[]){
 ic=Ice::initialize(argc,argv);
 Ice::ObjectPrx base = ic->stringToProxy("
     AudioSamplesService:default␣-h␣
     167.119.xxx.yyy␣-p␣10001");
 AudioSamplesServicePrx remoteService =
     AudioSamplesServicePrx::checkedCast(
     base);

 if(!remoteService)
  throw "Invalid␣proxy";

 cout << "processSamples()␣output:␣" <<
     remoteService->processSamples(img) <<
     endl;
}
```

The server application (`server.cpp`) implements the remote functions specified by the remote interface code. Below, it is schetched in the server code. Also, the it makes itself visible to the patient side by using the Ice function `createObjectAdapterWithEndpoints` that returns a remote handle `adapter` later added as a remote entity with `adapter->add`.

```

class AudioSamplesServiceI : public
    AudioSamplesService {
public:
 virtual std::string processSamples(const
     FloatSeq& img, const Ice::Current&);
 virtual std::string noise(const string& s,
      const Ice::Current&);
};

int
main(int argc, char* argv[])
{
 int status=0;
 Ice::CommunicatorPtr ic;
 //Platform system calls - Resource access

 ic=Ice::initialize(argc, argv);
 Ice::ObjectAdapterPtr adapter = ic->
     createObjectAdapterWithEndpoints ("
     audio_adapter",         "default␣-p␣
     10001");
 Ice::ObjectPtr object= new
     AudioSamplesServiceI;
 adapter -> add(object,ic->stringToIdentity
     ("AudioSamplesService"));
 adapter->activate();
 ic->waitForShutdown();

}
```

4.3 AMQP development

The programming of the audio system in AMQP to enable parallel audio transmission and processing is shown in figure 4.

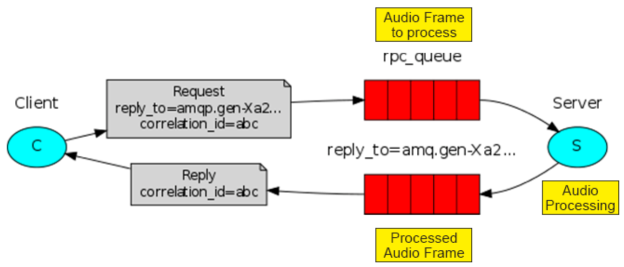

Figure 4: Architectural design of the audio system based on AMQP

Audio samples are sent from the client side to the server. Upon the arrival of an audio frame, it is processed by the server. Then, the server may either send an acknowledgement or reply audio media; this is configurable at initialization time. An `rpcClient` object is used to provide a `Call` method to start the communication by a remote message sending. The patient side software system is coded as follows by a simple call to `sendAudioSamples`.

```
var rpcClient = new RPCClient();
Console.WriteLine("[x] Requesting 
     process_samples(--Audio Sample--)");
var response = rpcClient.Call (rpcClient.
     sendAudioSamples());
Console.WriteLine("Elapsed_Frame={0}",sw.
     Elapsed);
Console.WriteLine (" [.] Got '{0}'",
     response);
rpcClient.Close();
```

The server side code is shown below. Audio is processed with `process_samples`. The response to the client is sent as an ACK with `BasicAck` and a `BasicPublish` through an *exchange*. This structure shows a round trip full-duplex communication scheme between the patients and the server side. Channel properties have to be checked to retreive the answers through the exchange.

```
try {
 string message = Encoding.UTF8.GetString(
      body);
 int[] n = ToIntArray(message, ';');
 response = process_samples(n).ToString();
}
catch(Exception e) {
 Console.WriteLine(" [.] " + e.Message);
 response = "";
}
finally {
 var responseBytes = Encoding.UTF8.GetBytes
      (response);
 channel.BasicPublish(exchange: "",
 channel.BasicAck(deliveryTag: ea.
      DeliveryTag,
 routingKey: props.ReplyTo,
 basicProperties           : replyProps,
 body: responseBytes);
 multiple: false);
}
```

Creating a callback queue for every RPC request is inefficient, but this is solved in a good way by creating a single callback queue per client. Still this may raise a design issue when receiving a response in that queue. In such a case, it is not clear to which request the response belongs to. That is when the *correlationId* property is used. It is set it to a unique value for every request. Later, when a message is received in the callback queue, this property is checked to match a response with a request. An unknown *correlationId* value, allows to safely discard the message as it does not belong to the requests.

Unknown messages are ignored in the callback queue, rather than failing with an error due to a possibility of a race condition on the server side.

In summary the RPC will works as follows. Upon client start up, an anonymous exclusive callback queue is created. The client side then sendsaudio samples as messages with two properties: *replyTo* set to the callback queue and *correlationId* that is set to a unique value for every request. Then, the request is sent to an *rpc_queue*. The server is waiting for audio samples in the queue. The patient side waits for data and acknowledgements on the callback queue. Upon arrival of messages the patient side checks the *correlationId* property; if it matches the value of the send audio message, it retreives the content.

5. CONTROL OVER THE PLATFORM RESOURCES

This section describes the mechanisms offered by both middleware technologies for applications to access the platform. Controling elapsed time and using timers as fault detection and recovery mechanisms are fundamental to some medical application types. Therefore, analyzing the mechanisms provided by specific middleware is needed for engineers to select the appropriate tools.

For developing the audio application, several operating system mechanisms and low level programming tunning have been used to have control over the execution. They are described in what follows.

5.1 Ice platform access

Several techniques have been used in the Ice application for managing execution.

Communications are monitored measuring the elapsed time with `clock_gettime` that uses high resolution clocks defined in POSIX such as `CLOCK_REALTIME` clock. Invocation time on is measured with the same clock to determine the performance of the link.

Mechanisms to prioritize the execution are also used. This enables the specification of patients of different priority depending on several factors such as patient health conditions or pay service. The operating system's real-time schedulers are used for this purpose (SCHED_RR, SCHED_FIFO) and function call `sched_setscheduler` and `sched_priority`. A real-time kernel (or patch to Linux) is needed to actually achieve priorization. Simultaneous requests from the patients are processed in parallel. Audio streams are assigned different processors using affinities.

5.2 AMQP

Most implementations of the protocol typically use programming languages that introduce a virtual machine, e.g., Java and its JVM or C# and Mono. Therefore, applications do not have direct access to the platform to have fine grain control over elapsed time calculations, etc. Language specific constructs used for AMQP implementation with C# have effect only inside the virtual machine process. These constructs are related to basic synchronization activities such as `Mutex`, `Monitor`, `Interlocked`, or `Stopwatch`. These constructions are managed by the virtual machine; although most of the time they are simply mapped to the OS system calls, their invocation experiments additional delays as the kernel has to schedule the virtual machine for execution, and the virtual machine itself has to schedule its own threads.

6. EXPERIMENTAL RESULTS

The execution scenario for the audio system for both platforms is similar; two patient sides transferring simultaneously audio media to a server side; the server sends back an acknowledgement to patient sides. Interfering load is synthetically produced to measure the stability of these technologies for the scenario. Load in both sides surpasses 90%. Figure 5 shows the comparison for both middleware technologies, whereas figure 6 presents differences with respect to the performance provided by different transport protocols.

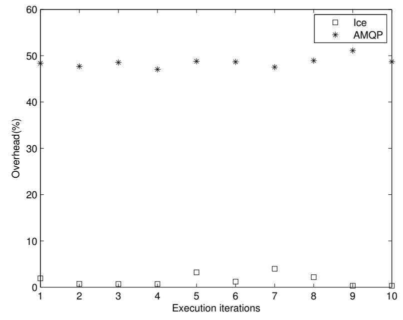

Figure 5: Experimental results: Ice ZeroC implementation vs AMQP RabbitMQ implementation

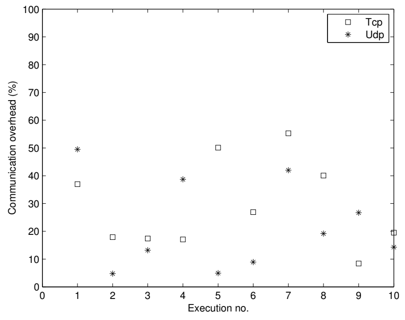

Figure 6: Different transports performance in Ice middleware

Ice middleware is the ZeroC implementation version 3.5, programmed in C++. Therefore, it provides the application with direct control over the resources, including the processor affinities, that impressively speeds up the execution at the server side. Ice run-time contains a set of threads that efficiently manage the communication functions between patient and server sides. The results show that the communication overhead is impressively smaller than the AMQP Rabbit technology [22]. Rabbit implementation uses a C# environment with a virtual machine, Mono, that provides a separate process for executing the application functions. Therefore, the application access to the platform resources is bounded by the execution of the virtual machine process that is scheduled by the kernel. As both platforms are not homogeneous, the overhead caused by the middleware communication protocols is shown in percentage, which allows to compare them: the overhead is the percentage of the round trip invocation time, measured at the patient side, for the same duration of the processing of the audio frames. Network effects are not analysed in the experiment as they are quite stable for an Ethernet link in the same subnet. Only the processing time of the middleware protocol stack in the individual machines is measured. The complete round trip time considers the processing of the packaged audio frames at the server side; this operation takes an average time of $1.55s$ in both platforms. With this value, these experiments can be reproduced for any platform and compared against these two middleware technologies.

7. CONCLUSIONS

An audio application in a medical scenario requires that the audio samples are processed at a continuous rate. These applications must be supported by a technologies that provide stable run-times, efficient communication, and support to achieve the required response times. Ice C++ enables control and management of the execution. However, RabbitMQ tests required a Mono client or compiling and executing the application. As RabbitMQ works as a virtual machine at the same level as the other applications of the operating system, the audio system cannot issue system calls in a direct way, which reduces its efficiency. After concluding the tests for both technologies, it has been observed that the resulting overheads are very different. This occurs mainly due to the inherent nature of each technology. It is concluded that there are open source middleware technologies that provide simple programming models, offering stable execution environments to applications. This is the case of both Ice and AMQP Rabbit implementations; they are good options to be considered by engineers as low cost solutions to implementing soft real-time medical systems.

8. REFERENCES

[1] Apache Software Foundation. *Jini^{TM} network technologies specification. Apache River v2.2.0.* `https://river.apache.org/doc/spec-index.html` (on-line). November 2013.

[2] J. Cano, M. García-Valls. *Scheduling component replacement for timely execution in dynamic systems.* Software: Practice and Experience, vol. 44(8), pp. 889-910. August 2014.

[3] N. Deakin. *JSR 343: Java^{TM} Message Service 2.0.* Oracle. March 2013.

[4] M. García Valls, T. Cucinotta, C. Lu. *Challenges in real-time virtualization and predictable cloud computing.* Journal of Systems Architecture, vol. 60(9). Oct 2014.

[5] M. García-Valls, D. Perez-Palacin, R. Mirandola. *Time-sensitive adaptation in CPS through run-time configuration generation and verification.* IEEE COMPSAC. Sweden. July 2014.

[6] M. García Valls, P. Basanta-Val. *Analyzing point–to–point DDS communication over desktop virtualization software.* Computer Standards & Interfaces 49, pp.11-21. January 2017.

[7] M. García-Valls, L. Fernández Villar, I. Rodríguez López. *iLAND: An enhanced middleware for real-time reconfiguration of service oriented distributed real-time systems.* IEEE Transactions on Industrial Informatics, vol. 9(1). 2013.

[8] M. García-Valls, P. Uriol-Resuela, F. Ibánez-Vázquez, P. Basanta-Val. *Low complexity reconfiguration for*

data-intensive service-oriented applications. Future Generation Computer Systems, vol.37. July 2014.

[9] M. García-Valls. *A Proposal for Cost-Effective Server Usage in CPS in the Presence of Dynamic Client Requests.* 19^{th} IEEE International Symposium on Real-Time Distributed Computing (ISORC), pp. 19-26. York , UK. May 2016.

[10] M. García-Valls, C.Calva-Urrego, A. Alonso, J. A. de la Puente. *Adjusting middleware knobs to suit CPS domains* 31^{st} Annual ACM Symposium on Applied Computing (SAC). Pisa, Italy. April 2016.

[11] M. García-Valls, P. Basanta-Val. *A real-time perspective of service composition: key concepts and some contributions*. Journal of Systems Architecture, vol. 59(10), pp. 1414âĂŞ1423. November 2013.

[12] ISO/IEC ITTF. *OASIS AMQP1.0 – Advanced Message Queuing Protocol (AMQP), v1.0*. 2014.

[13] K. Krishna. *Computer-Based Industrial Control*. https://books.google.com, (PHI Learning), 2010.

[14] S.M. Kuo, B.H. Lee, and W. Tian. *Real-Time Digital Signal Processing: Implementations and Applications*, Wiley, 2006.

[15] S. Kudrle, M. Proulx, P. Carrieres, M. Lopez. *Fingerprinting for Solving A/V Synchronization Issues within Broadcast Environments*. Motion Imaging Journal (SMPTE). 2011.

[16] A. Menychtas , D. Kyriazis , K. Tserpes, *Real-time reconfiguration for guaranteeing QoS provisioning levels in Grid environments*. Future Generation Computer Systems, vol. 25(7), pp. 779âĂŞ784. July 2009.

[17] S. Slichting, S. Polhsen. *An architecture for distributed systems of medical devices in high acuity environments. A Proposal for Standards Adoption*. Drager. 2014.

[18] X. Su, G. Swart, B. Goetz, B. Oliver, and P. Sandoz. *Changing engines in midstream: A Java stream computational model for big data processing*. In Proc. of VLDB Endowment, vol. 7(13), pp. 1343-1354. August 2014.

[19] J. O'Hara. *Toward a commodity enterprise middleware*, ACM Queue, vol. 5, pp. 48-âĂŞ55. 2007.

[20] *OMG.: Common Object Request Broker Architecture (CORBA) Specification, Version 3.1. Interfaces.* 2008.

[21] *OMG.: A Data Distribution Service for Real-time Systems Version 1.2. Real-Time Systems.* (2007)

[22] Pivotal software. *RabbitMQ. AMQP 0-9-1 Model Explained*. http: //www.rabbitmq.com/tutorials/amqp-concepts.html 2016.

[23] Sun Microsystems. *Java^{TM} Remote Method Invocation Specification*. Revision 1.7, Java^{TM} 2 SDK, standard edition, v1.3.0. December 1999.

[24] S. Vinoski. *Advanced Message Queuing Protocol*. IEEE Internet Computing, vol. 10, pp. 87âĂŞ89. 2006

[25] ZeroC Inc. *The Internet Communications Engine*. `https://zeroc.com/downloads/ice/3.5/` (on-line). 2016.