MIDDLEWARE TRENDS AND MARKET LEADERS 2011

A. Dworak, P. Charrue, F. Ehm, W. Sliwinski, M. Sobczak, CERN, Geneva, Switzerland

Abstract

The Controls Middleware (CMW) project was launched over ten years ago. Its main goal was to unify middleware solutions used to operate CERN accelerators. An important part of the project, the equipment access library RDA, was based on CORBA, an unquestionable standard at the time. RDA became an operational and critical part of the infrastructure, yet the demanding run-time environment revealed some shortcomings of the system. Accumulation of fixes and workarounds led to unnecessary complexity. RDA became difficult to maintain and to extend. CORBA proved to be rather a cumbersome product than a panacea. Fortunately, many new transport frameworks appeared since then. They boasted a better design and supported concepts that made them easy to use. Willing to profit from the new libraries, the CMW team updated user requirements and in their terms investigated eventual CORBA substitutes. The process consisted of several phases: a review of middleware solutions belonging to different categories (e.g. data-centric, object-, and message-oriented) and their applicability to a communication model in RDA; evaluation of several market recognized products and promising start-ups; prototyping of typical communication scenarios; testing the libraries against exceptional situations and errors; verifying that mandatory performance constraints were met. Thanks to the investigation performed the team have selected a few libraries that suit their needs better than CORBA. Further prototyping will select the best candidate.

CERN MIDDLEWARE

The Controls Middleware (CMW) project was launched at CERN over ten years ago. Its main goal was to unify middleware solutions used to operate CERN accelerators. Many software components were developed, among them the Remote Device Access (RDA) [1] library. The main responsibility of the library was to allow communication with servers that operate hardware sensors and actuators. The RDA design corresponds to the Accelerator Device Model [1] in which devices, named entities in the control system, can be controlled via properties. RDA implements this model in a distributed environment with devices residing in front-end servers that can run anywhere in the controls network. It provides a location-independent and reliable access to devices from control programs. By invoking the device access methods, clients can read, write, and subscribe to device property values. Currently over 4000 servers (processes) are deployed, which contain altogether almost 80,000 devices. In total the system gives access to more than 2,000,000 properties/IO points, on which clients may perform read/write operations or monitor their values. [2]

Present Implementation

From the beginning there were certain requirements [3] imposed on RDA that drove its implementation: relying only on standards; interoperability with the already existing communication infrastructure at CERN; portable on LynxOS with an old gcc v.2.95 compiler, Linux, Windows, HP-UX and AIX (only the first three are still supported; LynxOS is being eradicated); C/C++ and Java bindings for client/server libraries; request-reply and publish-subscribe operations on device data. Each call type should provide timeout settings and handling of communication errors. Moreover, complementary, centrally managed services like naming service, reservation service and access control should be supplied.

To facilitate development of the new library it was decided to base it on an already existing, mature product. CORBA [4] was a very popular middleware at that time and fulfilled all the requirements. Thus it was chosen as the communication layer. The C++ implementation was based on omniORB (currently 4.1.2,) and the Java implementation on JacORB (currently 2.2.4.) RDA library wrapped CORBA, hiding all its complexities and providing a simple to use API. The proposed solution was widely accepted and became an operational and critical part of the infrastructure.

Shortcomings of the System

Unfortunately, the demanding run-time environment revealed a few shortcomings of RDA. Accumulation of fixes and workarounds led to unnecessary complexity. Desire to deliver a better, more user-friendly solution led to a general review of the system. Discussions with library clients helped to identify several major issues, of which the most troublesome are the ones directly correlated with CORBA [5]. First, the CORBA standard is inherently huge and complex. Libraries that try to fully implement it have a major memory footprint. This is an issue especially for older front-end computers. It is well understood that RDA as a communication framework uses only a small fraction of the CORBA platform, but users still have to pay the full run-time price. On the other hand, libraries such as JacORB do not implement the full functionality. This leads to mismatches in behaviour of Java and C++ bindings. The struggle to support "asynchronous" operations on top of the synchronous calls leads to unnecessary complexity in the library code and design. Second, the way CORBA is used in RDA leads to multiple data conversions between different representations. This is both time consuming and leads to higher memory usage. Third, CORBA is based on the static Interface Definition Language (IDL), which is difficult to manage and evolve in large, complex environments such as CERN. Finally, the community supporting open-source implementations is shrinking.

Copyright © 2011 by the respective authors — cc Creative Commons Attribution 3.0 (CC BY 3.0)

There is a significant lack of new releases from the major implementations like JacORB, even if the major bugs have been identified and fixed a long time ago.

MIDDLEWARE EVALUATION

In view of a 1-year accelerator shutdown at CERN, starting end of 2012, there is a unique opportunity for introducing a major new version of RDA, which should solve all the limitations experienced with CORBA. Therefore, the CMW team launched the middleware review process, aiming at choosing a new, modern middleware library, to be used for the future version of RDA.

In addition to previously specified general requirements we expect that the new transport library provides:

- Consistent implementation for C++ and Java.
- Easy to trace peer-to-peer communication with reliable request/reply and publish/subscribe messaging patterns.
- Synchronous and asynchronous communication
- Quality of Service (QoS): timeout management, message queues and priorities, various thread management policies.
- Small library size, low memory and resource usage.
- Certain performance characteristics (described later)
- No, or only a few, external dependencies that can be linked with an application, preferably no need for additional services (e.g. brokers, global servers, daemons).
- Open source, with a license allowing to redistribute our product further; good documentation, and support from a large active community.
- Simple, easy to learn and use API.

The CMW team evaluated several market recognized middleware products. A short description of each product is provided below, including a general assessment and results of tests. Detailed performance results and other quantitative measurements are gathered and presented in the next paragraph. All opinions and criticism are based only on our knowledge and products evaluation.

In line with the requirements the following middleware standards and protocols were of no interest: XML-based protocols (e.g. SOAP, XMPP), Stomp, P2P (FastTrack, BitTorrent), MPI, MQTT (rsmb, Mosquitto) nor WebSphere MQ.

The Current Solution: omniORB/JacORB

CORBA is an object-oriented communication platform created by OMG. The standard defines the wire protocol and the IDL, which is used to specify object interfaces. It describes also mappings from IDL to several languages. The complexity of the communication process is hidden from the user, who cannot differentiate between a local and a remote call. The standard and chosen implementations are well documented. Unfortunately there are many shortcomings described in the previous paragraph. Also, the CORBA API is old-fashioned and heavy, thus it has a very steep learning curve and its community shrinks.

Evaluation of Ice

Ice [6] belongs to the object-oriented middleware category. It is conceptually very similar to CORBA, which is an advantage for those who already know it.

The product supports C++ and Java, and runs on Linux and Windows. Compilation on LynxOS fails due to the use of modern C++. It has a static type system and relies on separate specification files to describe interfaces and data structures. Apart from a request-reply model, Ice provides a publish-subscribe event distribution service called IceStorm. Full control over QoS and many tuning options are available. Performance wise Ice satisfies our needs. It uses a compact binary encoding that conserves bandwidth and is very efficient to marshal and unmarshal. Additionally protocol compression can be enabled. Sizes of statically compiled libraries and of binaries of a simple ping-pong server and client indicate a heavy use of global state that brings in the majority of Ice, no matter how much of it is actually used. On the other hand, the well designed API, modern and flexible IDL, easy to use language mappings, up-to-date documentation and a detailed tutorial are a big plus. The library is distributed with GPL license; sources are available for download.

Ice seems to be a very strong candidate due to its industrial presence and number of existing deployments. It also fulfils majority of our requirements.

Evaluation of Thrift

Thrift [7] belongs to the service-oriented middleware category, which means that the central notion in this system is that of remote services being accessed over the network.

The library supports C++/Java and runs on Linux/Windows. Compilation for LynxOS is problematic due to the use of modern C++ features. Thrift has a static type system and relies on separate specification files to describe the service interface and data structures. It supports simple request-reply communication in synchronous and asynchronous mode. It has a small memory footprint and fulfils the performance needs, but it is still an immature product with an incomplete implementation. Tutorial on the product webpage is empty and there is no documentation.

We decided to exclude Thrift from further investigation.

Evaluation of ZeroMQ

ZeroMQ [8] is a message-oriented middleware library, which resembles the standard Berkeley sockets. Because of supported communication patterns and various transports like in-process, inter-process, TCP and multicast it may be easily used as a concurrency framework.

The core of the library is written in C. Bindings for C++, Java (through JNI) and many more languages are supported. The library runs on most modern platforms.

Copyright © 2011 by the respective authors — cc Creative Commons Attribution 3.0 (CC BY 3.0)

With minor changes it is possible to run it on LynxOS. ZeroMQ has no type specification and does not know anything about the data a user sends. For this reason it has to be used with an external serializer. Because of similarities to the BSD sockets the API is familiar and easy to learn and use. In contrast to the BSD, the ZeroMQ API is more intuitive and user-friendly. Moreover, apart from simple socket send/recv calls, many complex communication patterns are implemented and ready to be used (e.g. request-reply, publish-subscribe, workload distribution). Users have full control over communication policies and QoS (synchronous or asynchronous communication, timeouts, high water marks). The library has a small memory footprint. To achieve the best possible performance it uses different protocols depending on the peers location (TCP, PGM multicast, IPC, inproc shared memory). Parallel protocols may be easily changed so an eventual upgrade from unicast to multicast is easy. The direct connection between the system parts results also in reduced maintenance costs as there is no need for brokers or daemons. A detailed documentation and broad, easy to follow tutorial are available on the product website. The project is under the LGPL license, with a large and active open source community. If needed, full commercial support may be obtained from iMatix, the authors of the product.

We consider ZeroMQ as one of the major candidates to replace CORBA.

Evaluation of YAMI4

YAMI4 [9] belongs to the message-oriented middleware category, in which communicating peers exchange messages between each other. The distribution is therefore explicit and seen in the user code.

The library supports C++/Java and runs on Linux/Windows. With small changes it is possible to compile it for LynxOS. YAMI4 has a dynamic type specification. Data structures (messages) are created dynamically without describing them with IDL. It is an inherently asynchronous communication system with support for request-reply and publish-subscribe over TCP. QoS may be configured through message priorities and timeouts. The library has a small memory footprint and, as our tests show, even if considerably slower than the statically typed products, it fulfils the performance needs. It is an open-source project under GPL, with a thorough documentation and a modern, intuitive API.

YAMI4 is already successfully used at CERN. Unfortunately, community behind the product is small.

Evaluation of the DDS Products

DDS [10] (Data Distribution Service) is an OMG standard, targeting real-time distributed systems. It belongs to the data-oriented middleware category, where the communicating parties declare their interest in a topic and the system takes care of delivery of only relevant data.

There are five wire-interoperable implementations of DDS. We evaluated the three most mature ones. All three products support C++ and Java languages, however due to use of modern C++ they do not support LynxOS out of the box. DDS has a static type system and relies on separate specification files to describe data structures. Compatibility of the generated code with the code generated from the CORBA IDL may be accomplished. Single-direction data flow is the most frequent use-case. It is possible to set up request-reply communication but this requires two symmetric channels. Because of the nature of the channels, this approach is not applicable for CMW, thus additional request-reply middleware would have to be used in parallel. DDS is an asynchronous system that supports many QoS settings, including message priorities. A nice additional feature is Dynamic Discovery, which allows a DDS application an automatic discovery and connection with another DDS application. This feature does not work for us as our network do not support multicast. The products are well documented, but the DDS API is neither easy to use nor compact. In fact, the multitude of settings and concepts provided by the standard is overwhelming and renders the products to be cumbersome and difficult to use.

Evaluation of OpenSplice DDS

OpenSpliceDDS [11] is the only DDS implementation that needs a separate daemon process on each node as individual user processes do not use the network services directly. The daemon is used for service discovery and for data transfer between nodes. Such a solution creates additional complexity, which should be avoided in CMW.

Evaluation of CoreDX DDS

CoreDX [12] is a small-footprint DDS implementation. Unfortunately, due to the licensing policy, further redistribution to third parties would be problematic.

Evaluation of RTI DDS

RTI [13] provides the most mature and widely adopted implementation of DDS. It is distributed with a number of useful tools for system monitoring and administration. As a research organization, CERN is eligible for a free of charge IRAD license and even access to the source code is available. On the other hand, the library size and simple binary programs are significant.

Evaluation of AMQP family

AMQP [14] is a wire-level protocol used for messaging. An AMQP system consists of a broker responsible for message routing between the communicating parties and a client library implementing the protocol. It does not provide any data model - only binary messages are supported. As AMQP is a broker system, implementation of request-response is cumbersome and almost two times slower than in a direct mode. Only recently, the first stable version of the protocol was released, but there is still no product that supports it. On the other hand, there are a few products using the previous, noncompliant versions of the protocol: Qpid v 0.10, OpenAMQ and RabbitMQ v 0.9, and

Copyright © 2011 by the respective authors — cc Creative Commons Attribution 3.0 (CC BY 3.0)

SwiftMQ v 0.8. The AMQP standard is still evolving and every new protocol version is not backward compatible. Moreover, the specification is still a work in progress and there is no clear indication on its future direction and support from industry [15]. Two products were evaluated, Qpid and OpenAMQ, however taking into account all the outstanding issues around AMQP, we decided to withdraw them from further investigations.

PERFORMANCE TESTS

The communication within CMW should be reliable and fast. Analysing the current usage statistics, it was estimated that the new transport over a GbE network, between a server on a new front-end (Inter Core 2 Duo, 1.5GHz, 1GB RAM, GbE) and a client running on a similar machine should handle approximately:

1) 4000msg/sec req-rep calls, payload = 4Bytes
2) 5msg/sec req-rep calls, payload = 10MBytes
3) publish 400 x 8B to 10 clients, in less than 100 msec
4) publish 30 x 8B to 10 clients, in less than 20 msec

For each candidate library all four scenarios were tested. The most interesting results were obtained from test 1 (see Figure 1), where the price for the YAMI4 dynamic model and additional hop through Qpid broker can be seen. A similar problem with IceStorm is revealed by test 3 (see Figure 2). That test also unveiled the brilliant, automatic message batching implemented in ZeroMQ.

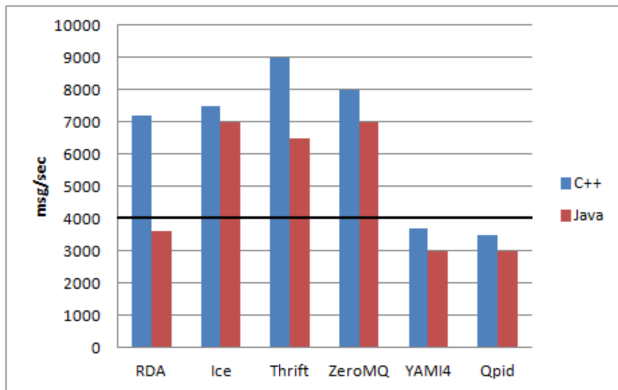

Figure 1: Test 1, a client talking to a C++ server.

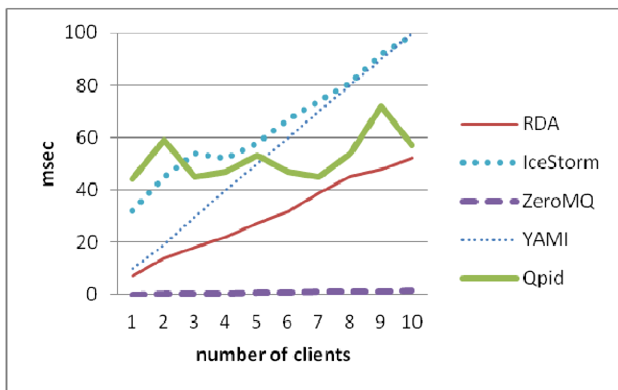

Figure 2: Test 3, pub-sub to a C++ server.

CONCLUSIONS

The paper presented several market recognized middleware products, evaluated according to the requirements of the CERN accelerator control system, as well as considering the product maturity and ease of use.

	patterns	QoS	resources	performance	user friendly	community	score
CORBA	✗	✓	✗	✓	✗	✗	2
Ice	✓	✓	✗	✓	✓	✓	5
Thrift	✗	✗	✓	✓	✗	✗	2
ZeroMQ	✓	✓	✓	✓	✓	✓	6
YAMI4	✓	✓	✓	✗	✓	✗	4
RTI	✗	✓	✗	✓	✗	✓	3
Qpid	✗	✓	✗	✗	✓	✓	3

Figure 3: Summary of evaluated middleware products.

The results are gathered in Figure 3. Three libraries were qualified for further prototyping: Ice, ZeroMQ and YAMI4. Based on prototyping the CMW team will select and adopt one of them for the future version of RDA.

REFERENCES

[1] N. Trofimov *et al.*, “Remote Device Access in the new CERN Accelerator Controls middleware”, ICALEPCS 2001, San Jose, California, 2001.
[2] Z. Zaharieva *et al.*, “Database Foundation for the Configuration Management of the CERN Accelerator Controls System”, ICALEPCS'11, Grenoble, France, October 2011.
[3] V. Bagiollini *et al.*, “CERN PS/SL Middleware Project, User Requirements Document”, CERN Note SL/99-16(CO), Issue 1 Revision 3, Geneva, Switzerland, August 1999.
[4] OMG CORBA http://www.corba.org/
[5] M. Henning, “The rise and fall of CORBA”, http://queue.acm.org/detail.cfm?id=1142044, 2006.
[6] ZeroC Ice: http://www.zeroc.com/
[7] Apache Thrift: http://thrift.apache.org/
[8] iMatix ZeroMQ : http://www.zeromq.org/
[9] Inspirel YAMI4: http://www.inspirel.com/yami4/
[10] OMG DDS: http://www.omgwiki.org/dds/
[11] OpenSplice: http://www.prismtech.com/opensplice
[12] CoreDX: http://www.twinoakscomputing.com/
[13] RTI: http://www.rti.com/
[14] AMQP: http://www.amqp.org/
[15] Pieter Hintjens, “What is wrong with AMQP”, http://www.imatix.com/articles:whats-wrong-with-amqp

Copyright © 2011 by the respective authors — cc Creative Commons Attribution 3.0 (CC BY 3.0)