LONGEVITY OF ACCELERATOR CONTROL SYSTEMS MIDDLEWARE

K. Žagar, Cosylab, Ljubljana, Slovenia

Abstract

Accelerators are designed to be in operation for several decades, and frequently even their construction alone takes a decade or more. Given the rapid rate of obsolescence of information technologies, it becomes a challenge how to choose the technologies that would stand the test of time, or at least make long-term support manageable. In this article, we focus on middleware: the glue that keeps inherently heterogeneous control system platforms able to interoperate with each another. We argue that whichever middleware technology is used, it is advisable to abstract it with simple, domain-specific APIs, whose implementation can change as the evolving performance requirements push the initial middleware choice beyond its limits of applicability.

INTRODUCTION

Middleware is infrastructural software that allows application-specific software components to interconnect. As middleware significantly simplifies development of distributed applications, control systems of large experimental facilities that involve more I/O (input-output) channels than a single computer can manage frequently adopt an existing middleware solution. In terms of the number of I/O channels, particle accelerators are probably one of the most demanding middleware applications.

For the purposes of this article, we extend the term "middleware" somewhat to include higher-layer frameworks that are already control-system specific. Thus, we consider frameworks such as EPICS [1], TANGO [2], TINE [3], OPC [4], LabVIEW [5] and others as middleware as well.

Middleware has a unique position in the control system architecture: most other software components interact directly with middleware. Therefore, once a middleware choice is made, it is difficult to change. Practically all software needs to be adapted, or more likely, rewritten – an effort which is significantly larger than can be made during a single maintenance shutdown and is more likely to take years (see for example the experience during the ESRF accelerator upgrade, [6]).

Contrast this to, say, replacing an obsolete I/O device with a newer model. This typically requires writing the device driver, and adapting it to its present interface through which the rest of the system interacts. If tested well outside the shutdown period, the site-wide replacement can be done in a matter of hours, or at most days – depending on the capability of management tools that come with the middleware.

Ideally, therefore, middleware would not need to be changed throughout the lifetime of the facility. Its *longevity* (or life expectancy) should thus be sufficiently long. By the very least, one should get a "life insurance" on the middleware: at the time of its adoption, take measures that will allow migration to another middleware implementation economically feasible.

In this paper, we first briefly explain the middleware concept, its benefits and pitfalls. Afterwards, we give some of the reasons why middleware might become obsolete at all. Then, we present some of the characteristics of middleware which we believe have the most impact on its longevity. Finally, we propose how to define architecture in such a way that change of middleware remains feasible.

BENEFITS (AND PITFALLS) OF MIDDLEWARE

In a sense, middleware is a data bus through which application's components share the data (see Figure 1). One of its principal benefits is *location transparency*, which hides most of the complexity of building applications that are distributed across several computers and require network communication to interact. Without middleware, application developers would need to resort to networking primitives such as sockets, and they would need to implement the data exchange protocol, including code for serialization of data structures to streams of bytes.

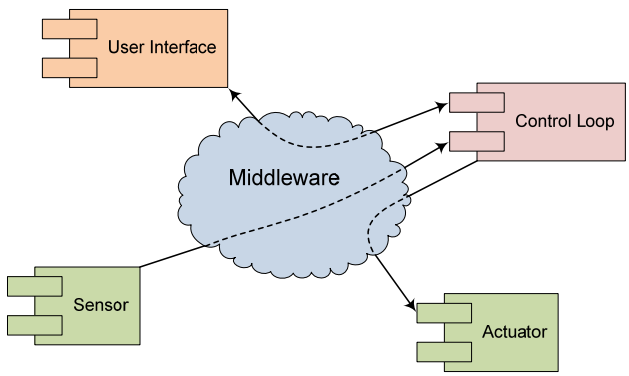

Figure 1: Middleware – the glue between application's software components.

Location transparency implies a step further – not only is development of distributed system simplified, one can easily switch from a monolithic, single-process deployment to a distributed one, without changing the application code.

But this benefit does not come without a significant pitfall which is frequently neglected: in a distributed system, a much wider spectrum of failures can occur. What is a simple method invocation in a single-process application can now fail due to reasons such as non-existence of the invocation target, network failure, invocation target process' crash, etc. Thus a good middleware will also provide facilities to handling faults

in a manner that does not require failure management considerations to "pollute" most of the application's code.

Another very significant benefit of middleware is that it allows interoperability of application's components not only across the network, but also among different hardware architectures, operating systems, and even computer languages.

REASONS FOR MIDDLEWARE OBSOLESCENCE

The reasons why middleware is prone to facing obsolescence as time passes is the same as with the rest of software:

- **Requirements change**, and the middleware no longer fulfils them. For example, an upgrade at the facility might require streaming experimental data at high data rates to dozens of consumers, but middleware does not allow for efficient multicasting.
- **Hardware and/or operating system evolve**, and middleware is no longer compatible. For example, existing hardware becomes obsolete and stock items are no longer available, therefore it must be updated. However, the update requires operating system to be upgraded as well, because the old operating system does not have all required drivers. But, the middleware does not work well with the new operating system...
- Middleware offering significantly higher level of development efficiency becomes available, and developers/maintainers opt for a change.

MIDDLEWARE LONGEVITY CHARACTERISTICS

Open Source

Because source code of open sourced middleware is readily available, the staff at experimental facilities can make slight modifications to account for the changing requirements and hardware/operating system environment. Also, with open source software it is easier to troubleshoot difficulties, as white-box inspection techniques can be applied. Thus, ceteris paribus, open source software is more likely to be long-lived than an equivalent closed source package.

We note that not all software that is open source is also available for the general public to modify. Such example is the Sun's Java library, the sources of which are readily available with the Java Development Kit distribution, but is not allowed to be modified.

Proprietary

Proprietary software can be purchased off-the-shelf. Due to economy of scale, the price of licenses and first few years of support are relatively low. Also, the solution is available practically immediately, as it is already developed. Furthermore, in cases where the software has a large install base, the software is likely to be reliable as it has been thoroughly tested by many users.

However, there are several risks that need to be considered:

- It is unlikely that the off-the-shelf software will satisfy all the present and future requirements of the experimental facility.
- There is possibility of a vendor lock-in: as the vendor owns a monopoly of the particular software, it might abuse its position and raise the price of support/licenses in the future.
- The vendor might not be around throughout the life-time of the experimental facility, or it might decide to cease supporting the software. This risk can be effectively mitigated with an escrow agreement which allows the user to obtain the source code, shall the vendor be unable to continue to support it.
- Only the vendor can modify the software to fit the needs of experimental facility, at their own pace. This risk can be mitigated with a support agreement, however support agreements spanning several decades are a risky proposition for the vendor, which might result in a high price for the user.

Given the mentioned benefits and risks, there are subsystems of experimental facilities that benefit from using proprietary software, for example:

- PLC-based safety and machine protection systems. E.g., at CERN, such systems are integrated with the UNICOS framework [7].
- National Instruments' LabVIEW, which can be used to implement control systems of entire experiments. E.g., the PHELIX experiment at GSI [8].

Interoperable Open Standard

Some middleware is standardized through an open standardization process. The standard typically prescribes the API as well as the wire protocol.

Because of standardized API, middleware implementation that conforms to the standard can be replaced with another implementation, without having to make any modification to the application code.

The standardized wire protocol allows applications that use different middleware implementations to communicate with one another (interoperability).

Some control systems infrastructures benefit from the open standard significantly, in particular those based on *Common Object Request Broker Architecture* (CORBA), which is standardized by the *Object Management Group* (OMG). An example of such a control system is the *ALMA Common Software* (ACS) [9], where CORBA implementations *OmniORB*, *JacORB* and *TAO* enable interoperability of Python, Java and C++ processes, respectively. Also, ACS was able to change one implementation for another during its lifetime, because support for the initial CORBA implementation ceased to exist.

Availability of Support

Long term support is crucial for longevity, for changing requirements and environment may eventually require making modifications to the middleware.

For open source packages, support is frequently offered by the community behind the package. Such is the example with EPICS and ACS, where support is offered through publicly available mailing lists (EPICS tech-talk and ACS discuss, respectively). However, as members of the community are primarily responsible for other tasks than supporting the middleware, therefore this sort of support cannot be considered guaranteed.

Apart from community support, commercial support is frequently available. For example, several companies offer guaranteed support for EPICS (Alceli Consulting Cosylab and Observatory Sciences).

Economy of Scale

Middleware that has many users will likely outlive the middleware with a smaller user community. This correlation is valid both for proprietary as well as community-developed middleware. In the first case, the company developing the middleware will generate larger revenues due to a large number of users, allowing for more support manpower. In the second case, the users themselves contribute to the middleware.

Simplicity

Complex middleware tends to be difficult to program against, repealing developers from using it and looking for more convenient alternatives.

Usually, complex middleware also has a large code-base due to numerous features it supports. Consequently, its memory footprint is larger, performance is worse, and more maintenance effort is required.

Therefore, middleware should be as simple as possible, and yet no simpler, as oversimplifications can lead to loss of important functionality.

Maturity

Characteristics of immature software are that its API changes frequently, and that every release brings about numerous defects (bugs). For middleware, this is particularly problematic, as all application-level software uses the middleware, and might either need to change due to middleware API's changes, or be rendered unstable due to a faulty release.

OBSOLESCENCE-PRONE ARCHITECTURE

In this section, we outline two architectural approaches that extend the lifetime of applications based on a particular middleware.

When having an opportunity to develop applications from scratch, it is a good idea to introduce an abstraction layer that hides the underlying middleware's APIs. If the middleware needs to be replaced, only the implementation of the abstraction layer needs to be adjusted, and applications need not change.

More likely scenario is one where applications written with "new" and "old" middleware need to co-exist. To summarize [12], there are three approaches how this can be achieved:

- Gateways – processes linked against both middleware libraries, capable of translating from one to another. This approach requires no modification of applications, but offers least performance and robustness.
- Client-side "plugs" – writing client applications against an abstract API. Examples of such architectures are the *XAL framework* used at SNS [10], and *Data Access Layer* used at DESY and at GSI [11].
- Server-side "translators" – server processes exposing data via several middlewares, giving clients a choice which middleware to use.

CONCLUSION

In this paper, we have outlined several aspects of middleware that are correlated with its longevity. Long-lived middleware would be available in open source (or be proprietary but based on an interoperable open standard), the support would be available – both from user community as well as commercially, it would have many existing users, it would exhibit simplicity, and its maturity would be proven.

REFERENCES

[1] "EPICS – Experimental Physics and Industrial Control System"; http://www.aps.anl.gov/epics.
[2] "TANGO – TACO Next Generation Objects", http://www.tango-controls.org/.
[3] "TINE – Three-fold Integrated Networking Environement", http://tine.desy.de.
[4] "OPC – OLE for Process Control", http://www.opcfoundation.org/.
[5] National Instruments, "NI LabVIEW"; http://www.ni.com/labview.
[6] J. Meyer et al., "Upgrading the ESRF Accelerator Control System after 10 Years of Operation", ICALEPCS'03.
[7] Ph. Gayet, R. Barillere, "UNICOS a framework to build industry like control systems", ICALEPCS'05.
[8] H. Brand et al, "The PHELIX Control System Based on UML Design Level Programming in LabVIEW", ICALEPCS'03.
[9] G. Raffi, G. Chiozzi, B. Glendenning, "The ALMA Common Software (ACS) as a Basis for a Distributed Software Development", Astronomical Data Analysis Software and Systems XI, ASP Conference Proceedings, Vol. 281, p. 103.
[10] J. Galambos et al, "XAL Application Programming Framework", ICALEPCS'03.
[11] I. Kriznar et al, "Beyond Abeans", ICALEPCS'07
[12] P. Duval et al, "The Babylonization of Control Systems Part II – The Rise of the Fallen Tower", ICALEPCS'03