WESAS'00 Position Paper
Eric M. Dashofy
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697
edashofy@ics.uci.edu

The Role of Middleware in Software Architectures

Architecture-based software development has shown great promise in increasing the flexibility, adaptability, and reusability of software systems. A popular definition of a software architecture partitions a system into three key elements: components, connectors, and configurations. The connectors in a software architecture play an important role in determining how flexible and adaptable a software system is. A system with only fixed, static connectors cannot be changed at runtime to allow the addition of new components. Connectors that only connect components in a single language or that run in a single environment can constrain the implementation language and platform of components in a system. As such, building diverse, standards-compliant multi-language and multi-platform connectors allows software architects to greatly expand the capabilities of their systems.

Single-process, single-machine connectors have been shown to provide a measure of dynamic adaptability for software systems. For instance, the developers of ArchStudio, the C2 design environment, have used the Java language's dynamic code loading capability and C2's notion of explicit connectors to dynamically load, attach, unload, and remove components on the fly during runtime. However, these changes occur within a single process, on one machine. Many additional dimensions of flexibility and adaptability lie in the confluence of *middleware* technologies and software architectural styles. Middleware is software that facilitates the communication of components across language, process, and machine boundaries. Among the many COTS middleware packages in use today in the computing world, several variations exist. First, each middleware technology has its own notion of what a "software component" is. CORBA and RMI view a software component as a single object in an object-oriented programming language. For Polylith (a software bus from the University of Maryland), a software component is a UNIX process. Second, different middleware technologies have varying platform and language support. CORBA ORBs are only required to support one language and platform, but may support many of both. Java RMI supports only one language, but will work on any platform supporting Java. Microsoft COM supports many languages, but only one platform (Windows). Third, different middleware technologies have different methods for inter-component communication. CORBA, RMI, and COM use remote procedure calls. Polylith uses a "shared" message bus. Message queue-based middleware like MQ series and MSMQ use individual message queues. Fourth, middleware packages differ in how much dynamism they allow

at runtime. Some middleware packages facilitate loading of in-process components at run-time, others allow change only at the process-level, and still others require that the system configuration be determined beforehand, allowing no run-time change.

[DMT99] offers an initial exploration of the use of middleware in the C2 architectural style's explicit software connectors. However, there is still quite a bit of work to be done. First, the use of middleware will have to be examined in the context of other architectural styles. The trade-offs between middleware and the architectural styles themselves will have to be addressed. For instance, it may be possible to implement a distributed pipe-and-filter architecture with an RPC-based package like a CORBA ORB, but it may be extremely inefficient compared to a message-bus package.

The additional dimensions of dynamism introduced by middleware are also important, but relatively unexplored. Whereas a given architectural style's ability to add and remove components at run-time may be well-understood within a single process framework, the ability to add and remove new processes and new machines to the style presents new challenges. These challenges include failure semantics, change semantics, visibility, and performance. In the area of failure semantics, a lot can go wrong when middleware becomes part of a software system. In a single-process system, programmers assume that a procedure call (a simple, implicit type of software connector) will always succeed. Control will transfer to the called procedure and the parameters will be passed on the stack appropriately. When a procedure call becomes a remote procedure call, failure of the underlying middleware, the network, and the inability to marshal parameters can all cause the call to fail. Furthermore, in systems that involve "hostile" environments or unreliable connections such as wireless networks, the failures may only be transient. Dealing with such failures in a uniform, unobtrusive manner without greatly disturbing the underlying architectural framework is an area of future work. In a related area, middleware modifies the change semantics of a system. Adding, removing, or replacing an existing component in a running system may be vastly different when middleware is involved. New methods may have to be developed to determine how to save the state of a component so it can be replaced with a newer version, or how to stop the message flow through the middleware to a component so it can be replaced without messages being lost. Depending on the middleware, services may have to be built to handle these cases, or the middleware itself could provide these services. In the area of visibility, the configuration of a system is more difficult to determine when more than one process or machine is involved. Determining what a system looks like (what components are in the system and how they are connected) is not too difficult in a single-process system. The use of a small "architecture manager" that is responsible for loading and connecting components can assist with this greatly. However, in a multi-process, multi-machine system, such "architecture managers" must coordinate and communicate to determine the current state of a system. As such, an originally

small part of the system has itself become a distributed system. Maintaining effective visibility of a distributed architecture is, thus, another area of future research. Finally, performance has to be taken into account. Traditionally, real-time systems depend on tight control of the underlying hardware and operating system to provide performance and priority guarantees. However, in a distributed system, performance guarantees will have to include the performance of many machines and an underlying network. Real-time middleware has been built, including Lockheed Martin's HARDpack, a real-time fault-tolerant CORBA ORB. However, the trade-offs between real-time software architectures and real-time middleware is not well-understood, and represents an area for future investigation.

Evaluating middleware-enabled architectures introduces even more challenges. Because distribution can reduce the visibility of software, doing architectural constraint checking can be more difficult in a distributed architecture. Checking the validity of interfaces and connections between components may also be difficult when the calling component and the called component reside on different machines or are written in different languages. Evaluating the performance of a middleware-enabled architecture is complicated because it increases the involvement of external factors like the performance of the underlying network and the operating system's process scheduler. The additional dimensions of dynamism presented by middleware create new criteria for evaluating whether a component can be added, removed, or replaced in the system that do not exist in a single-process implementation. All of these evaluation metrics represent unexplored areas for future work.

The use of middleware in software architectures presents new abilities and challenges to software architects. It enables multi-process, multi-language development, and the ability to control change on a level above that of a single component—adding a set of components or a small sub-application to a running system. These new abilities, however, come with a price. Understanding the trade-offs between middleware technologies and developing evaluation metrics, tool support, and architectural frameworks for distributed architectures will require additional work and consideration.

References:

[DMT99] E. M. Dashofy, N. Medvidovic, and R. N. Taylor, "Using Off-the-Shelf Middleware in Distributed Software Architectures." *Proceedings of ICSE'99*, Los Angeles, CA, 1999.