International Journal of Software Engineering
and Knowledge Engineering
Vol. 13, No. 4 (2003) 367–393

World Scientific
www.worldscientific.com

THE ROLE OF MIDDLEWARE IN ARCHITECTURE-BASED SOFTWARE DEVELOPMENT

NENAD MEDVIDOVIC*
Computer Science Department, University of Southern California,
Los Angeles, CA 90089, USA
neno@usc.edu

ERIC M. DASHOFY† and RICHARD N. TAYLOR‡
School of Information and Computer Science,
Department of Informatics, University of California, Irvine, Irvine, CA 92697, USA
† *edashofy@ics.uci.edu*
‡ *taylor@ics.uci.edu*

Software architectures promote development focused on modular functional building blocks (components), their interconnections (configurations), and their interactions (connectors). Since architecture-level components often contain complex functionality, it is reasonable to expect that their interactions will be complex as well. Middleware technologies such as CORBA, COM, and RMI provide a set of predefined services for enabling component composition and interaction. However, the potential role of such services in the implementations of software architectures is not well understood. In practice, middleware can resolve various types of component heterogeneity — across platform and language boundaries, for instance — but also can induce unwanted architectural constraints on application development. We present an approach in which components communicate through architecture-level software connectors that are implemented using middleware. This approach preserves the properties of the architecture-level connectors while leveraging the beneficial capabilities of the underlying middleware. We have implemented this approach in the context of a component- and message-based architectural style called C2 and demonstrated its utility in the context of several diverse applications. We argue that our approach provides a systematic and reasonable way to bridge the gap between architecture-level connectors and implementation-level middleware packages.

Keywords: Software architecture; connectors; middleware.

1. Introduction

Research into the field of software architecture has yielded a plethora of architectural styles, design notations, analysis techniques, and software tools for supporting the systematic design of large software systems out of components and connectors. The components in such systems tend to be heterogeneous and complex. That is,

*Corresponding author.

not only do they implement complex functional parts of the system, they may also be implemented by different organizations, written in different programming languages, run on different platforms, and have divergent expectations about the properties of systems of which they are a part. This leads to the well-known problems of component integration and architectural mismatch. Being able to connect software components in spite of such heterogeneity and then reason about their interconnections is vital for successful system integration.

Most software architecture research has focused on modeling and creating systems with explicit software connectors. These connectors are first-class software entities and facilitate the communication among two or more components. At the architectural level, connectors are high level abstractions that prescribe properties of component interconnections. Having these prescriptions available at design time facilitates modeling and analysis of the software system's design. Depending on the nature of the application, software connectors may be simple (a UNIX pipe or a local message bus) or complex (flexible publish/subscribe connectors across machine boundaries). These connectors are often part of a larger architectural style — a set of design constraints and patterns that elicit well-understood, beneficial system properties [35]. Unfortunately, to date, few approaches to architecture-driven development have explored how to implement connectors that match the architectural properties specified at design-time.

Implementation-time connection of heterogeneous software components has traditionally been the domain of middleware [12]. Different kinds of component heterogeneity can be (partially) resolved by off-the-shelf middleware solutions like CORBA [30], COM [33, 37], and JMS [20]. Middleware can help system integrators and architects to integrate components across platform, language, and machine boundaries. However, using middleware often induces certain architectural constraints on a system that may not be desirable [11]. For instance, using RPC-based middleware like CORBA implies that all components must communicate using RPC. This can have widespread effects on the design and analysis of any CORBA-based system. The properties of middleware may or may not match the properties of a connector specified in a software system's architecture, leading to mismatch between a software system's implementation and its design.

The relationship between architecture and middleware and their respective shortcomings suggest the possibility of coupling architecture modeling and analysis approaches with middleware technologies in order to get "the best of both worlds". Given that architectures are intended to describe systems at a high-level of abstraction, directly refining an architectural model into a design or implementation may not be possible. One reason is that the decision space rapidly expands with the decrease in abstraction levels: at the design level, constructs such as classes with attributes, operations, and associations, instances of objects collaborating in a scenario, and so forth, are identified; the implementation further requires the selection and instantiation of specific data structures and algorithms, interoperation with existing libraries, deployment of modules across process and machine boundaries,

and so forth. One proposed solution to this problem has been to provide mechanisms for refining an architectural model into its implementation via a sequence of intermediate models [4, 22, 28, 29]. However, the resulting approaches have had to trade off the engineer's confidence in the fidelity of a lower-level model to the higher-level one against the practicality of the adopted technique [25]. Furthermore, to a large extent, the existing refinement approaches have failed to take advantage of a growing body of existing (implemented) components that may be reusable "as is".

This paper pursues another strategy. We view software connectors as pieces of software that provide a service that connects components. This service has well-defined and well-understood properties, as specified at the architectural level. A middleware solution, or a combination of middlewares, can be used to facilitate implementation of this connector. Thus, the connector's properties influence the choice of middleware, rather than the middleware's characteristics affecting the properties of the connector. For example, a connector that uses message-based communication will probably be easier to implement using message-based middleware than using RPC-based middleware (although it is possible to use RPC-based middleware for this purpose, as we will show). The middleware used to implement a connector can also be influenced by other factors such as cost, ease of deployment, and implementation-level concerns like programming languages or platforms. An RPC-based connector that is used to connect Java components suggests that RMI is a better middleware than, say, COM.

Thus, we hypothesize that component interconnections can be designed at the architectural level and then facilitated by middleware, rather than having the choice of middleware constrain the connections between components. We have conducted a series of case studies to validate our hypothesis. A specific architectural style, C2, has been used as the basis for this investigation [24, 36]. In evaluating our approach, we learned several valuable lessons about integrating middleware into software connectors. Our results are promising and indicate that a successful marriage of architecture- and middleware-based techniques and technologies is indeed possible.

The rest of this paper is organized as follows: Section 2 discusses background material related to connectors and middleware. Section 3 discusses our approach in detail. Section 4 describes the applications we built using our approach and the lessons we learned from building these applications. Section 5 addresses general lessons learned and future directions for this work. Section 6 finishes the paper, covering the conclusions from our work.

2. Background

Exploring the relationship between software architecture-level connectors and middleware requires an understanding of existing architectural models for software connectors and the capabilities of modern middleware systems.

2.1. *Connectors in software architecture research*

The key role of connectors in architecture-based software development has been accepted by the majority of the software architecture community. For example, this is reflected in connectors becoming a part of the "core ontology" in the ACME architecture interchange language [34]. However, current architecture research is characterized by inconsistent approaches to fulfilling this key role of connectors. Three projects representative of the state of the practice are Wright [6], UniCon [34], and Rapide [22].

Wright is an architecture description language (ADL) whose particular focus is formally specifying protocols of interaction among components in an architecture. To this end, it employs a subset of communicating sequential processes (CSP) [15]. Given an architectural specification, Wright is able to determine the interaction characteristics of components communicating through any given connector, e.g., whether they will deadlock. However, Wright does not provide any support for the (correct) implementation of connectors.

UniCon, on the other hand, focuses on implementing connectors. To that end, it supports a predefined set of connectors: pipe, file I/O, procedure call(s), data access(es) and remote procedure call(s). UniCon's shortcoming is that it supports a limited set of connectors. Several of the connectors UniCon currently supports are simple and their implementation is either already provided by the chosen underlying programming language or is otherwise trivial. UniCon provides an elaborate mechanism and accompanying process for specifying new connector types with more complex protocols. However, it is unclear how or whether this mechanism can be used to incorporate any of the OTS middleware technologies discussed in Sec. 1.

Rapide is an ADL whose accompanying toolset provides extensive modeling, analysis, simulation, and code generation capabilities. However, Rapide does not model connectors as first-class entities, but rather specifies them in-line. This limits their reusability and renders their verification more difficult, as each connection must be analyzed individually. Implementation strategies and guidelines are thus required for each individual connector, rather than each connector type.

There is, therefore, a need for an approach where powerful and extensible connector modeling formalisms are coupled with connector implementation support and architecture simulation and code generation. This is a complex task. Our hypothesis is that implementing connectors with these properties can be made easier by building upon existing middleware technologies.

2.2. *Middleware*

A full taxonomy of modern middleware systems is far beyond the scope of this paper [12]. However, certain broad classifications of middleware can be made. For instance, middleware packages resolve various types of heterogeneity, as follows:

Platform Heterogeneity:

Middleware can allow communication among components running on different platforms. Many CORBA ORBs, for instance, have compatible implementations that run on various flavors of UNIX, Windows, MacOS, etc. Java-based middleware like RMI [1] and JMS also allow this, but they do this by leveraging the portability of the underlying virtual machine.

Language Heterogeneity:

Middleware can allow communication among components written in different programming languages. Microsoft's COM, for example, allows communication among components written in Visual Basic, C++, and other Microsoft languages. Contrast this with, say, RMI, in which all components must be written in Java.

Connectivity Heterogeneity:

Research middleware packages such as the QuO extensions to CORBA [38] explicitly address quality of service (QoS) and can help to resolve differences in aspects of connectivity like bandwidth and reliability. For instance, QoS-enabled middleware for media delivery can downsample audio or video so a 10 mbit stream is viewable by someone with a 56 kbit connection. Middleware with the ability to store-and-forward information is useful for components with unreliable network connectivity to other components or components that are nomadic.

Surprisingly, many popular industrial middleware packages assume a great deal of homogeneity of components as well, for both business-related and technical reasons. A single middleware technology will generally assume that all components use one type of invocation, for instance. Explicit invocation is well-supported by RPC-based middleware like CORBA, COM, and RMI. Implicit invocation using messages is well-supported by message-oriented middleware (MOM) like WebSphere MQ [17], Microsoft's MSMQ [16], myriad JMS implementations [5, 20], or research-oriented MOMs like JEDI [9] or Siena [8]. Support for other types of invocation are possible but clumsy in these packages; synchronous RPC can be built from messages, CORBA's event-service appears like an asynchronous event-notification service but in reality is little more than the Observer pattern [13] reified as a "CORBAservice". Certain middleware packages are built to support the business needs of their creators rather than at the expense of flexibility; COM's support for platforms other than Windows is practically nonexistent; RMI is not interoperable with components written in languages other than Java.

Middleware packages also vary across more specific, feature-oriented dimensions, for example:

Structural Dynamism:

Different middleware packages have varying levels of support for the addition and removal of components at run-time. Applications written in Polylith [32], for example, must have their structure specified completely in advance, while applications that use message-oriented middleware can establish and destroy connections as they run.

Performance:

Middleware for common desktop or business applications is generally optimized for these types of applications, but certain classes of applications such as flight control software or medical software may have strict real-time requirements. Middleware like HARDPack [21] and TAO [3] offer special features for building soft- and hard- real-time connectors.

Reliability and Fault Tolerance:

Synchronous and explicit invocation middleware like CORBA usually require that all calls succeed or an error is reported, usually through a programming-language exception. Asynchronous or implicit-invocation middleware can generally be configured in multiple modes (best effort, at-most-once, reliable with timeouts, and so on) depending on the needs of the application.

One drawback of using almost any modern middleware packages is the extent to which aspects of the middleware permeate the code of the components that use it. Consider a system built using CORBA. Component interfaces will be specified in IDL, have its data types influenced by the IDL type system, discover each other using the CORBA name service, and so on. Because of this, it is almost impossible to swap CORBA for a different middleware (RMI or JMS, for example) without significant recoding of components. This is a direct result of the choice of middleware influencing the design of a system, instead of the other way around.

3. Approach

Any approach to building software architecture-level connectors using middleware will depend heavily on the semantics of the architectural style being used. This section discusses the C2 style, which we used as the basis for our investigation, and how we applied our approach to build middleware-enabled connectors in that style.

3.1. *Overview of the C2 style*

We have chosen the C2 architectural style [36] as a foundation upon which to initially explore the issues of integrating middleware with software architectures. Basic constraints of the style are as follows:

- Components must assume that they do not share an address space with other components.
- All communication among components is performed using messages, which are independent data structures that do not contain direct pointers to data structures residing in any component.
- Components are assumed to run concurrently; that is, they may not assume that they share a thread of control with any other component.
- Components and connectors in the style have two distinct interfaces known as "top" and "bottom" interfaces. Components and connectors can send and receive

messages on both interfaces. Messages traveling "upward" are known as requests; messages traveling "downward" are notifications. Components may make assumptions about what services are provided by components above them but may not make assumptions about anything below them.

- A component may be connected to at most one connector on its top interface and one connector on its bottom interface. A connector may be connected to zero or more components or connectors on either interface. This induces a layered design on C2 systems.

An example of C2 architecture is depicted in Fig. 1.

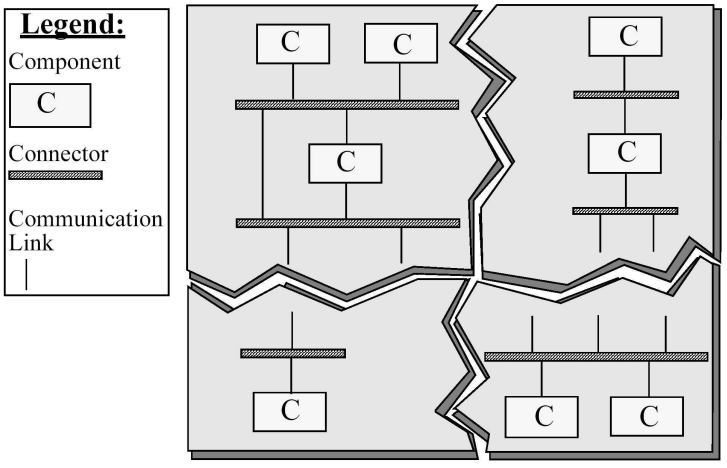

Fig. 1. Example C2 architecture. Jagged lines represent the parts of the architecture not shown.

The C2 style is a good fit for this task for several reasons. C2 has an explicit notion of software connectors as first-class entities that handle component interactions. The style prescribes certain basic properties of all C2 connectors: they are message-based, two-way multicast buses. Messages arriving on a connector's bottom interface are routed to components attached to the connector's top interface, and vice-versa. C2 connectors can also perform optional tasks such as filtering, routing, simple message transformation, and dynamic connection and disconnection of components, many of which are also typically provided by various middleware packages. The style is well-suited to a distributed setting as no assumption of shared memory or thread of control is made by any set of components. This allows us to leverage and experiment with the networking capabilities of middleware technologies. C2 supports a paradigm for composing systems in which components may be running in a distributed, heterogeneous environment, architectures may be changed dynamically, multiple users may be interacting with the system, multiple user interface toolkits may be employed, and multiple media types may be involved.

3.1.1. *Architecture frameworks*

Many architecture-based development approaches and architectural styles are focused on supporting system design only, and lack support for implementing systems in the given approach or style. This is a key cause of architectural drift, in which a system's implementation diverges from its original architectural design. Maintaining a software system's design in its implementation has many benefits, among them the ability to reason about or analyze properties of a design and be able to induce those properties in the implemented system. Unfortunately, architectural styles like C2 rely on constructs not usually found in object-oriented programming languages. Software libraries called *architecture frameworks* [27] bridge this gap. An architecture framework is a piece of software that provides constructs and functionality specified by an architectural style in the context of a given programming language or environment. It could be said, for instance, that the standard I/O library in the C programming language is the architecture framework for the pipe-and-filter style because it provides constructs (standard input and output bytestreams) as well as methods for accessing them (functions like read(), write(), etc.) that are not native or inherent in the C language.

More complex architectural styles require more complex architecture frameworks. We have implemented architecture frameworks for supporting development of C2-style applications in several different programming languages on several different platforms. To date, we have built C2 frameworks in C, C++, Ada, Java, and Python. How constructs from the style are implemented in the given programming language depends largely on the features of the language. For instance, the C implementation of the framework represents components as software libraries with predefined entrypoints. In an object-oriented programming language like Java or C++, component implementations are generally class libraries containing a distinguished class inheriting from an abstract *Component* base class provided by the framework. Desired framework characteristics can influence how the framework is built, as well. We have several C2 frameworks built in Java for different purposes: for example, one is a lightweight framework with fixed threading and message queuing policies, one is a heavyweight framework where arbitrary threading and queuing policies can be plugged in at compile-time.

The C2 architecture frameworks also influence how software connectors are implemented. The object-oriented frameworks generally provide some sort of abstract notion of a connector, such as a base class or an interface, that can be extended or implemented to build a connector. Our frameworks also generally provide a trivial implementation of an in-process two-way broadcast bus as a basis for making in-process connections among homogeneous components. They also provide a way for components and connectors to send and receive messages in the context of the local programming language. Usually, sending a message is done through a sendMessage() method call that returns immediately. Components and connectors receive messages either through callbacks or polling, depending on the framework.

Thus, a component communicates only by sending and receiving messages to and from adjacent connectors. The connector's responsibility is to communicate those messages to the appropriate set of connected components, even if those components are on different machines or platforms, or are written in different programming languages. This is where middleware's functionality is needed.

3.2. *Encapsulating middleware in connectors*

Our approach to integrating architecture-based software design and middleware is to retain design knowledge at implementation time and facilitate implementation of that design using middleware. This can be accomplished by encapsulating middleware inside implementations of software connectors [10, 26].

Middleware systems are implementation-level efforts to resolve various types of component heterogeneity. In doing so, however, they may impose constraints or assumptions on development that are contrary to the wishes of a software system's designers. For example, consider implementing a C2 connector in CORBA or any other RPC-based middleware. In this case, the middleware can allow two components written in different languages or on different machines to communicate, but it also forces the components to communicate in a synchronous request-response point-to-point fashion, which is not consistent with the asynchronous, one-way, multicast semantics of a C2 connector. In theory, a C2 connector based on CORBA should appear (to attached components) just like an in-process C2 connector or a connector facilitated by any other type of middleware. That is, the service of asynchronous message routing provided by the connector should be constant, even if the method by which the service is provided changes. This is already accepted in component-based development; separation of a component's interface(s) from its implementation is considered fundamental to good design. Similarly, we believe that a connector's interface should be separate from its implementation, no matter what middleware is used to facilitate that implementation. Somewhat surprisingly, this does not hold true for most connectors/middleware in use today. As discussed above, most middleware packages today require components to be coded with a specific middleware package in mind.

This perspective suggests our general approach: a software architect designs an application in the most appropriate and intuitive way, selecting or adapting an appropriate architectural style. The architect selects the needed off-the-shelf (OTS) components or designs new ones, then lays out the topology of components and connectors (as guided by constraints of the architectural style). When the components and their topology are known, one or more middleware platforms that are suitable for implementing the connectors are chosen and used in the implementation. If middleware-based connectors with the appropriate characteristics have already been developed (in a previous project, or as part of the architecture framework, for example), they can be reused easily.

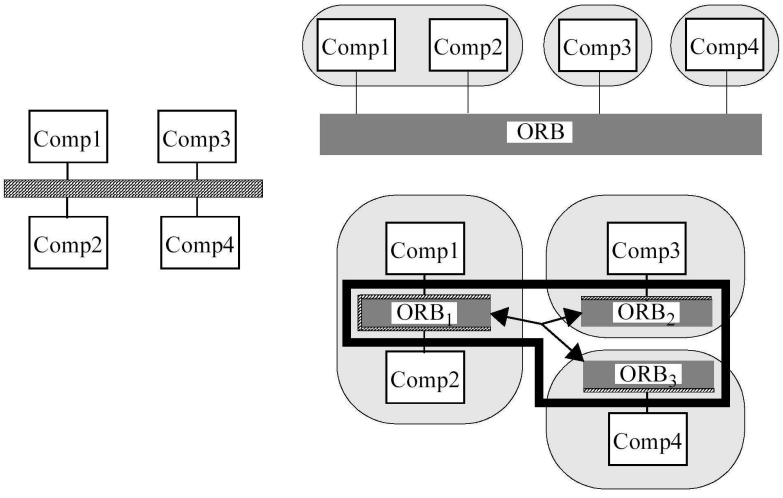

Fig. 2. Realizing a software architecture (left) using a middleware technology (top-right) and an explicit, middleware-enabled software connector (bottom-right).

A simple example that illustrates this strategy is shown in Fig. 2. A conceptual architecture of a system is shown on the left. In this case, the C2 style mandates that information flow only up and down through the connector (e.g., *Comp1* and *Comp3* cannot directly interact, while *Comp1* and *Comp2* can). Assume we ignore connectors in our implementation and want to implement the architecture with components bound to a given ORB-based middleware, distributing the implementation over three locations. The top-right diagram depicts the resulting solution: the single ORB ensures the cross-machine interaction of its attached components, but not the topological and interaction constraints imposed by the style.

Contrast this with our approach, depicted on the bottom-right of Fig. 2. This is a more principled way of integrating architectures and middleware. Recall that we keep connectors as an explicit part of a system's implementation infrastructure. Each component only exchanges information with a connector to which it is directly attached; in turn, the connector will (re)package that information and deliver it to its recipients using one or more middleware technologies. Each such "middleware-enabled" connector exposes the same interface (to attached components) as an in-process connector; it changes the underlying mechanism for marshalling and delivering messages, but externally appears unchanged. Note that, unlike the "middleware-only" solution shown in the top-right diagram, the bottom-right diagram of Fig. 2 also preserves the topological and stylistic constraints of the application. The middleware-enabled connectors are independent of other connectors in the system, and can optimize aspects of communication where middleware is not needed, such as the in-process connection linking *Comp1* and *Comp2*.

Maintaining a connector's interface and semantics allows much more flexibility in application development and deployment. For instance, if it were decided later

that *Comp3* and *Comp4* should indeed run in the same process, it would be possible to make this change by simply reconfiguring the connector. The component code would not have to be changed at all. The middleware package could be swapped out similarly (for one that cost less or had better performance for example), also without affecting component code.

Depending on the components being connected, a given middleware package may or may not be a "close fit" — that is, it may or may not fit all the properties of the connector perfectly. In this case, developers have three choices:

Adapt Middleware with Glue Code:

If the middleware's features are close to the required connector properties, the least expensive or difficult route will be to write glue code that adapts the middleware to the necessary features of the connector. For instance, a best-effort messaging middleware can be adapted for use in a reliable messaging connector with the use of retries and acknowledgements. Event-based middleware can be adapted for use in a synchronous request-response connector via the use of synchronization and blocking constructs.

Combine Middleware Solutions:

When one middleware solution is not sufficient, it may be possible to combine middleware technologies to achieve the desired result. For example, consider connecting a Java 1.4 application to an off-the-shelf COM-based component. Microsoft's Java Virtual Machine (JVM) provides middleware for interfacing Java 1.1 to COM, but does not support Java 1.4. Sun's JVM supports Java 1.4, but does not support COM. However, both JVMs support RMI. Therefore, it is possible to build a Java 1.4 → (RMI) → Java 1.1 → (COM) → COM bridge by combining these middleware solutions.

Build New Middleware:

If the properties required of a given connector are radically different from any available middleware solution, the cheapest alternative may be to build new, special-purpose middleware for inclusion in a connector. This is generally very expensive, but may be the least costly alternative in this case.

Encapsulating middleware in connectors in this way increases the flexibility an architect has in designing a system. A system can be initially designed as one large architecture, regardless of where process, language, or communication-method boundaries will end up when the system is implemented. As these decisions are made (when specific off-the-shelf components are selected, for example, or the deployment hardware for the system is chosen), connector properties will become more and more concrete.

A key insight in encapsulating middleware in connectors that cross process boundaries is that a software connector does not need to be implemented as a single code artifact; in fact, the connector itself is simply a service that may span process or even machine boundaries. In Fig. 2, a single connector in the software's abstract architecture description is actually implemented by three separate software

artifacts running in different processes; together, these three segments make up a single "virtual connector" with the same overall semantics as the architecture-level connector.

In practice, it is not always possible to maintain all the semantics of an in-process connector when a middleware-based connector is used instead. Notably, the failure semantics for distributed or multi-process connectors are richer than those for in-process connectors. An in-process connector, for instance, will likely never drop messages or invocations. Multi-process or distributed connectors, on the other hand, will have to deal with permanent and transient network failures, process failures, and dropped messages. We believe that, for most architectures, decisions on where process and machine boundaries will go will happen relatively early in the architectural design process. These types of failures occur whenever any kind of distributed systems middleware is used in a system. Therefore, as long as these failure semantics are anticipated early in architectural design, the architecture will still retain the flexibility to swap out middleware or reorganize connectors later. Dealing with connector failures in a reasonable way results in more robust and reliable architectures, and should be part of any serious architectural design in the first place.

3.3. *Application to the C2 style*

Deciding how to break up a multi-process connector and still maintain the semantics of the connector as a whole depends on the architectural style in question. Figure 3 depicts two methods that can be used in the C2 style. The top of the figure shows a technique we call "lateral connection" in which two or more connector segments (in this case, *Conn1* and *Conn2*) are "split" horizontally and internally connected laterally. Components may be attached to the top or bottom of any segment. In-process messages from top to bottom, or vice versa, are passed through without going through any middleware. The bottom of the figure shows a variant of this technique called "vertical connection" in which a connector is "split" vertically. Because there are no valid in-process message paths in this strategy, all messages pass through the middleware. Lateral connection is more flexible, but slightly more complicated than vertical connection for two reasons: First, lateral connections must process in-process messages separately. Second, optimization techniques should ideally be used to minimize network usage. For instance, in the top half of Fig. 3, notifications sent downward from *Comp1* should not be sent across the network to *Conn2* because there are no receivers below *Conn2*.

We implemented multi-process middleware-enabled C2 connectors in the C++ and Java frameworks using many different middleware packages: ILU [2], VisiBroker CORBA [7], RMI [1], Polylith [32], Q [23], and E-Speak [14]. The resulting connectors are arbitrarily composable to support any deployment profile. The motivation for such a composition is that different middleware technologies may have unique benefits. By combining multiple such technologies in a single application,

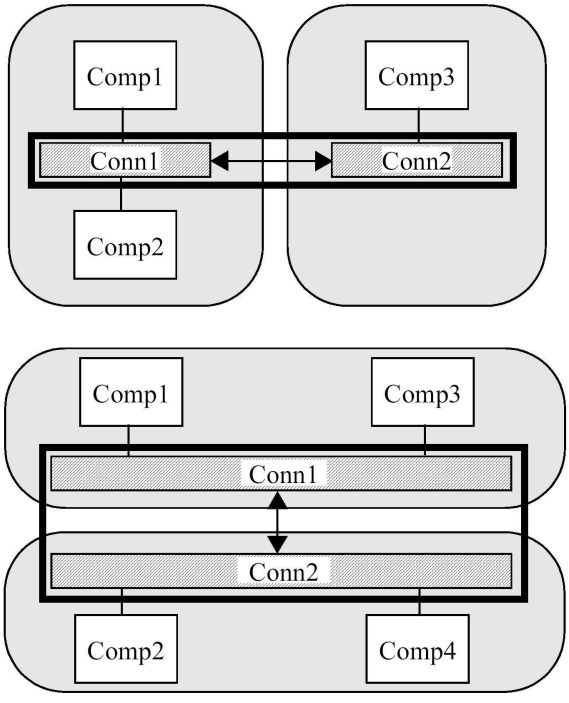

Fig. 3. Connectors as a primary vehicle for interprocess communication. A single conceptual connector can be "broken up" vertically (top) or horizontally (bottom) for this purpose. Shaded ovals represent process boundaries. Each connector encapsulates an ORB (elided for simplicity).

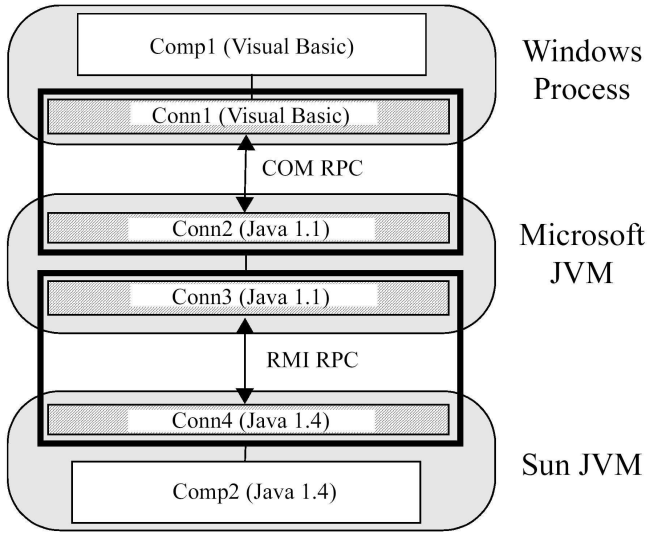

Fig. 4. Using multiple middleware packages to obtain the benefits of both. RMI and COM are both used here to integrate a Visual Basic and Java 1.4 component.

the application can potentially obtain the benefits of all of them. For instance, a middleware technology that supports multiple platforms but only a single language, such as RMI, could be combined with one that supports multiple languages but a single platform, such as Q, to create an application that supports both multiple languages and multiple platforms. Figure 4 shows an instance where both RMI and COM are used to integrate a Java 1.4 component with a Visual Basic component, as described in Sec. 3.2. In fact, we used this exact strategy to integrate Microsoft Visio with a Java-based development environment. Certainly, there are performance and memory penalties to integrating multiple middleware solutions in this way, but the alternative was to write custom middleware ourselves or purchase a COM-to-Java bridge for hundreds or thousands of dollars [18, 19]. For a prototype integration and proof of concept, the RMI+COM integration was sufficient. If later we chose to replace this with custom middleware or an off-the-shelf bridge, however, we would not have to change component code.

3.4. *Lessons learned*

Building middleware-enabled C2 connectors taught us a lot about the process of integrating middleware into software connectors. Lessons included:

Mapping Procedure Calls to Messages:

C2 connectors communicate using one-way, asynchronous messages. Many of the middleware packages we evaluated (ILU, RMI, Q, and CORBA) communicate, by default, using synchronous request-response RPC. We do this by having one connector segment simply call a public method on the other(s), passing the message as a parameter in the call. This preserves the asynchronous, non-blocking semantics of C2 connectors when each connector segment runs in its own thread of control. However, if all components and connectors in an architecture share a thread of control, a cross-process message send can hold up the whole architecture until the called connector has received the message. However, in practice this rarely occurs. Message-oriented middleware, or asynchronous calls like CORBA one-way calls, are a better fit for message passing communication.

Discovery of Other Connector Segments:

In a multi-segment connector, each segment must be able to locate other segments running in other processes. Most middleware packages we evaluated include some sort of nameserver that allows lookup of other segments by pre-defined names passed to the connector segments as configuration parameters. Notable exceptions include Q and E-Speak. In Q, no nameserver is provided, so we built a simple nameserver that provided this functionality. E-Speak uses service lookup rather than lookup-by-name, since it is envisioned as a service-matching middleware. This can allow a little more flexibility in how applications can configure themselves, as connector segments can locate other connector segments based on service characteristics rather than simply by name. An interesting question is how to represent

the nameserver architecturally, since the middleware nameservers we encountered did not fit within the constraints of the C2 style. In the end, we chose to treat the nameserver as a piece of infrastructure that was an internal part of the distributed connector; therefore, it did not need to be modeled in C2. Because of the dynamic nature of C2 applications (components and processes coming and going at runtime) we believe that a more dynamic discovery protocol, using peer-to-peer discovery techniques [31], is more applicable to C2-style architectures than a database-like nameserver.

Establishing Connections between Connector Segments:

In an in-process connector, the whole connector is instantiated along with the rest of the components in the application. In a multi-process virtual connector, order and time of instantiation must be taken into account. Some of the middleware packages we used, like ILU, CORBA, RMI, and E-Speak, are decidedly client-server oriented. This means that server objects are expected to come online before client objects, and that clients connect to servers at some later time. If a server object is not online when a client object attempts to connect, an error is thrown. This is generally not a problem in C2 architectures when vertical connections (bottom half of Fig. 3) are used because of the dependency relationships of the C2 style. Components and connectors "above" are server objects and components and connectors "below" are clients. As long as the "above" portions of the architecture are instantiated before "below" portions, the system will function normally. In the case of lateral connection (top half of Fig. 3), the situation is slightly more complicated since each segment of the connector must be both a client and a server. In this case, connector segments must implement additional code to be able to look each other up and connect to each other regardless of the startup order of the application processes.

Unintended Consequences:

Depending on the properties of the underlying middleware, interesting capabilities may come to the application without writing additional code. For instance, middleware that allows objects and processes to come, go, connect and disconnect from each other at runtime like RMI or CORBA increases application dynamism. Creating new connector segments at runtime that will look up and connect to existing connector segments can change the topology, and therefore the behavior of the application. This dynamism is achieved without writing any sort of additional code to connect components in-process. In an architectural style that encourages runtime change like C2, this is a beneficial, if unintentional, consequence of using certain kinds of middleware.

4. Evaluation

After building several middleware-enabled connectors, we used these connectors to create several multi-process, multi-platform, and multi-language versions of sample software applications.

4.1. *Video game*

One application we used as a platform for evaluating our work is a C2 version of the video game KLAX, a falling-tiles game originally developed by Atari Corp. KLAX was chosen because game play imposes time constraints on the application, bringing performance issues to the forefront. The application's architecture is depicted in Fig. 5. The components that make up the KLAX game can be divided into three logical groups. At the top of the architecture are the components that encapsulate the game's state. The game state components respond to request messages and emit notifications of internal state changes. Notification messages are directed to both the game logic components and the artist components. The game logic components request changes of game state in accordance with game rules and interpret the change notifications to determine the state of the game in progress. The artist components also receive notifications of game state changes, causing them to update their depictions. Each artist maintains the state of a set of abstract graphical objects which, when modified, send state change notifications in the hope that a lower-level graphics component will render them on the screen. *GraphicsBinding* receives all notifications about the state of the artists' graphical objects and translates them into calls to a window system. User events, such as a key press, are translated by *GraphicsBinding* into requests to the artist components.

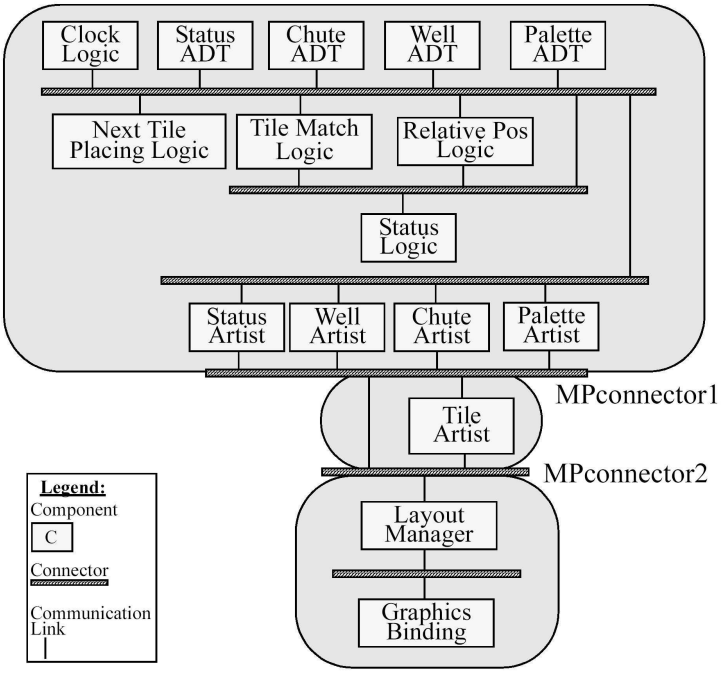

Fig. 5. Architecture and deployment profile for KLAX. Shaded ovals represent process boundaries.

We used several deployment profiles to examine the issues in using middleware technologies to implement architectures; one such profile is shown in Fig. 5. Two KLAX implementations were built using the C++ and Java versions of the C2 architectural framework. A variation of the architecture was also used as the basis of a distributed, multi-player KLAX application in which players compete via a remote *GameServer*. The *GameServer*, in turn, notifies the appropriate players of the changes to their states in response to their opponent's action.

Performance of the different implementations of KLAX easily exceeds human reaction time if the *ClockLogic* component is set to use short time intervals. Although we have not yet tried to optimize performance, benchmarks indicate that the C++ framework can send 1200 simple messages per second when sending and receiving components are in the same process, with the Java framework being somewhat slower. In single-player KLAX, a keystroke typically causes 10 to 30 message sends, and a tick of the clock typically causes 3 to 20 message sends. The efficiency of message exchange across process and/or machine boundaries is a function of the network bandwidth and the underlying mechanism (i.e., middleware) used to implement the given inter-process/machine connector.

4.2. *Software development environment*

DRADEL is an environment that supports modeling, analysis, evolution, and implementation of C2-style architectures. It has also served as a platform for experimenting with middleware-enabled connectors. Just like the application architectures it is built to support, the architecture of DRADEL itself, shown in Fig. 6, adheres to C2 style rules. The environment is built using the Java C2 implementation framework.

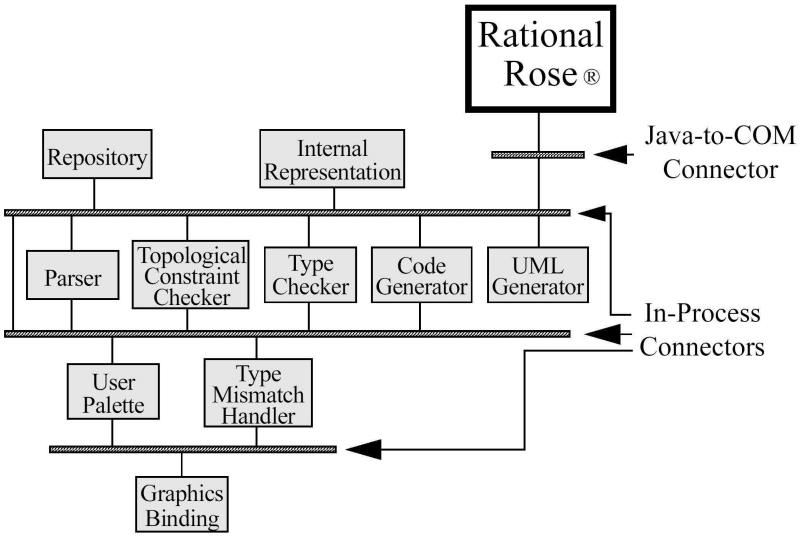

Fig. 6. Architecture of the DRADEL environment.

The *User Palette*, *Type Mismatch Handler*, and *Graphics Binding* components from Fig. 6 provide a graphical front end for the environment. The *Repository* component stores application architecture models. The *Parser* receives via C2 messages a specification of an architecture, parses it, and requests that the *Internal Representation* component check its consistency and store it. Once the specification of an architecture is parsed and its internal consistency ensured, the *Topological Constraint Checker*, *Type Checker*, *Code Generator*, and *UML Generator* components are notified of it. *Topological Constraint Checker* ensures adherence to the topological rules of C2 discussed in Sec. 3.1. *Type Checker* analyzes architectures to establish conformance among interacting components. The *Code Generator* component generates an implementation skeleton for the modeled architecture on top of the Java C2 framework.

This base architecture has been deployed in several ways across traditional (i.e., desktop) and mobile (i.e., hand-held) platforms using middleware-enabled connectors. Since our example in the next section focuses on mobile environments, we will briefly discuss another enhancement of DRADEL, depicted in the upper-right portion of Fig. 6. In order to provide developers with support for refining architectures into designs and implementations that are potentially independent of the C2 framework, we have evolved DRADEL with the *UML Generator* component. *UML Generator* implements a set of rules for transforming an architectural model into a UML model, and issues a series of requests for building such a model. These requests are routed to a connector that encapsulates Microsoft's Java-to-COM middleware. In turn, the connector bridges DRADEL's topmost connector and Rational Rose, a COTS environment for UML-based software development. Rose uses the information contained in the requests sent by *UML Generator* to build a UML model corresponding to a C2 architectural model of an application.

4.3. *Personnel deployment application*

The third application we have extensively coupled with our middleware-enabled connectors enables distributed deployment of personnel in situations such as military crises and search-and-rescue efforts. The specific instance of the application depicted in Fig. 7 addresses military Troops Deployment and battle Simulations (TDS). A computer at *Headquarters* gathers information from the field and displays the current battlefield status: the locations of friendly and enemy troops, vehicles, and obstacles such as mine fields. The headquarters computer is networked via secure links to a set of PDAs used by *Commanders* in the field. The commander PDAs can connect directly to each other and to a large number of *Soldier* PDAs. Each commander is capable of controlling his own part of the battlefield: deploying troops, analyzing the deployment strategy, transferring troops between commanders, and so on. In case the *Headquarters* device fails, a designated *Commander* assumes the role of *Headquarters*. Soldiers can only view the segment of the battlefield in which they are located, receive direct orders from the commanders,

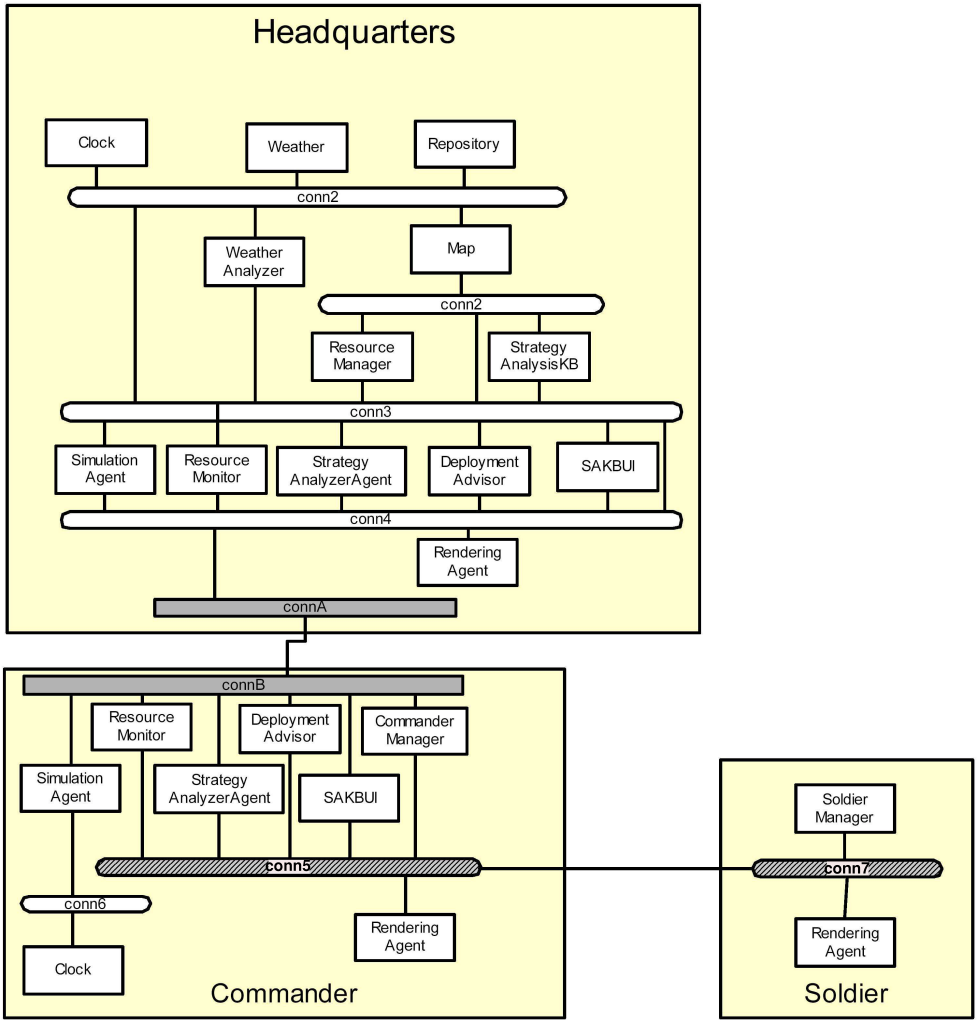

Fig. 7. TDS application architecture distributed across three devices. The four middleware-enabled connectors are highlighted. The connectors bridging the Commander and Soldier devices are both in-process and inter-process connectors.

and report their status. Figure 7 shows a deployment with single *Headquarters*, *Commander*, and *Soldier* devices.

The TDS application has provided an effective platform for investigating a number of the concepts discussed in this paper. A version of TDS has been designed, analyzed, implemented, deployed, and dynamically evolved using the C2 style and C2's architectural implementation framework. TDS is implemented in four dialects of two programming languages: Java JVM, Java 2 Micro Edition (J2ME), C++, and Embedded Visual C++ (EVC++). The largest configuration of TDS constructed to date has been deployed onto 105 devices with application size totalling over

13 MB. Each device contains multiple components and connectors. The devices on which TDS has been deployed are of several different types (Palm Pilot Vx and VIIx, Compaq iPAQ, HP Jornada, NEC MobilePro, Sun Ultra, PC), running four OSs (PalmOS, WindowsCE, Windows 2000, and Solaris). In turn, this allowed us to investigate the suitability of multiple middleware solutions for use in concert in this setting. Because the Headquarters subsystem runs on a PC, we had several choices for middleware (e.g. RMI and CORBA). The Commander subsystem, running on a Compaq iPAQ using JDK 1.1.8, limits our choice of middleware to RMI. As discussed in Sec. 4.4 below, the connection to the Soldier subsystem, running on a Palm Pilot, required us to implement a small bit of custom lightweight middleware using raw sockets due to space constraints.

4.4. *Lessons learned*

Our construction and integration of a number of different middleware implementations with the three applications described above revealed that the choice of middleware can have unintended consequences. Some of them were beneficial, while others were detrimental to the given application. Below we highlight those issues that we believe to be independent of our chosen architectural style, application domains, or applications.

Performance Issues (*Jitter*):

For a large class of applications, issues such as processing speed and communication latency are every bit as critical as functional correctness. For example, KLAX's need for consistently high performance, to support real-time gameplay, put stress on many of the used middleware implementations. This was most obvious with the Polylith integration into the C++ architecture framework. Polylith uses basic UNIX IPC constructs to exchange data among processes and the UNIX process scheduler schedules timeslices for each process in the system. On the Solaris machines we used for testing, the scheduler gave out large timeslices to each process in the application. This resulted in "bursty" message exchanges between the processes. The end-result was that messages would be processed in short bursts with delays in between, essentially resulting in an unplayable game. For applications such as DRADEL and even TDS, where constant streams of messages are not critical, this would not have been an issue, but it was an insurmountable problem in a real-time video game. This experience stresses that understanding the performance and behavioral characteristics of both the application and the chosen middleware is very important.

Performance Issues (*Mismatch*):

Certain applications, such as KLAX and TDS, are characterized by high message volumes. Deploying and running different parts of such an application on machines with different speeds can cause performance mismatches: the faster process may produce events at a higher rate than the slower process is able to consume. If this is

a temporary mismatch (e.g., one of the processes was running garbage collection) the effect on the application may be negligible. However, if this mismatch is permanent, the application may become unusable as messages back up in the faster process. This is certainly the case with a video game such as KLAX. There are two possible solutions to this problem: components may be designed to deal with lost messages and the IPC connectors enhanced to drop messages when their internal queues reach a certain length; alternatively, the components generating the offending messages may be instructed to do so at a lower frequency. For example, KLAX's Clock component can be modified to emit fewer ticks, slowing the game down so all processes have ample time to process messages. Of course, such a modification will impact the application's functionality. This is another indication that understanding the performance characteristics of an application and the various deployment platforms is critical.

Application Size (*Memory Footprint*):

A growing class of software applications is intended to run on devices with limited computational resources, such as PDAs or mobile telephones. An issue of particular importance in such environments is the amount of available dynamic memory. For example, the Palm Pilot, used for the deployment of the Soldier functionality in the TDS application, has between 64 KB and 256 KB of dynamic heap memory. Such a constraint eliminates from consideration almost all of the existing middleware solutions, which routinely require several megabytes of memory; even the implementation of RMI targeted at J2ME requires 90 KB of memory, which may be unacceptable on Palm Pilots. Indeed, we were forced to implement the segment of the distributed connector running on the Soldier device in Fig. 7 using only "bare" network sockets.

Interoperability Issues (*Cross-Platform Portability*):

Middleware solutions often make assumptions about the computing platform(s) on which they will be installed and used. For example, RMI assumes and depends upon the presence of Java, while Polylith is targeted at Unix environments. This frequently limits the applicability of a middleware solution to a specific set of platforms. Another limiting factor is the assumption that certain capabilities will be readily provided by the hardware device and/or operating system on which the middleware is to be installed. While this may have been a realistic assumption in traditional (i.e., desktop) environments, it need not be the case for an emerging class of novel and possibly experimental hardware platforms (e.g., mobile phones and PDAs), or for specialized computing environments targeted at such platforms. Thus, for example, Sun's KVM implementation of J2ME does not support server-side sockets, rendering unusable any middleware solution, including Sun's own RMI, that relies on such support. This is an indication of the impact that the emerging, heterogeneous computing platforms are likely to have on the usability of off-the-shelf middleware.

Interoperability Issues (*Available APIs*):

One reason for using standard middleware solutions is that they will inexpensively enable interoperability with third-party components. This was the case with DRADEL's integration with Rational Rose, shown in Figure 6. However, the employed middleware solution and/or third-party component may make certain assumptions, reified in the middleware's and/or third-party component's respective APIs, which are inconsistent with their desired use. For example, the connector enhanced with the Java-to-COM middleware and the integrated version of Rose provided effective support for communicating from DRADEL's *UML Generator* component to Rose, by issuing requests. However, there was no suitable way of relaying any notifications originating inside Rose back to DRADEL, rendering the communication uni-directional. Remedying such a situation would have required finer-grain control over the middleware than is usually provided. The user of the off-the-shelf middleware is then left with the choice of dealing with an imperfect system or custom-building the desired functionality, which may be a very expensive proposition.

Unintended Features:

Not all unforeseen consequences of using a standard middleware solution are necessarily negative. For example, in the deployment of the KLAX application shown in Fig. 5, the *Layout Manager* and *Graphics Binding* components are responsible for rendering the game display on screen. Depending on the implementation of the relevant connector segments and the middleware platform used, creating a second copy of the two components' container process and attaching it to the middleware-enabled *MPconnector2* results in a second window that can be used to observe a game in progress from another machine, without changing *any* application code. This potentially useful feature of the game was achieved "for free" because of the flexibility of the employed middleware packages.

5. Critical Evaluation and Future Work

Overall, we believe our approach and evaluation have demonstrated that it is feasible to encapsulate middleware in connectors, allowing system architects to choose the properties of connectors first, and find middleware that allows them to implement those connectors later. The success of our approach thus far motivates several future research goals.

First, we have applied our approach only to C2-style architectures so far. In the future, we would like to experiment with different architectural styles. Our approach is based on two principal aspects of C2: explicit connectors and event-based communication. C2's use of explicit connectors allows us to easily separate the interface of the connector from its implementation, allowing us to more elegantly encapsulate middleware within them. C2's exclusive use of event-based communication means that connectors must basically implement one simple primitive, namely one-way transmission of an independent data object. Our approach is independent of many

other aspects and constraints of C2, such as the layered topology, the default use of broadcast buses as connectors, and the separation of request and notification messages. Therefore, we believe that our approach is applicable as-is to any architectural style that uses explicit connectors and message-passing as its only form of communication.

Styles that use explicit connectors with other forms of communication, such as remote procedure calls (RPC), should also be possible in our approach, but we have not evaluated them directly. Still, we have shown how RPC-based middleware can be used to implement event-based connectors, and it is possible to construct the request-response mechanism of RPC using one-way events and thread blocking. Application of our approach to connectors that use non-event-based communication is an area of future work for us.

Styles that use implicit connectors (some forms of shared memory, procedure calls, and so on) are slightly more difficult to integrate into an approach such as ours, since there is no obvious separation between the connectors' interfaces and their implementations. In this case, we believe that programming language-level tools such as compilers and preprocessors must be co-opted to intercept uses of these implicit connectors and redirect them to alternate, middleware-based implementations. This is also a future direction for us.

Performance issues highlighted above such as jitter and mismatch revealed an important general issue: the ability to state architecture-level properties of connectors in advance of implementing them is very important for this approach. As the properties of a connector become better understood, it also becomes easier to effectively choose middleware to implement that connector. Some properties for distributed connectors can be chosen for the convenience of component developers (events vs. RPC, for example). When implementing distributed connectors, however, some properties are induced by the network itself. For example, it is possible to create an in-process connector that never drops messages. This is strictly not possible in a distributed setting: network failures can usually break a connector, even if reliability-increasing techniques such as retries and alternate routing are employed. Properties of the underlying network, such as throughput and latency, are hard lower-limits on the throughput and latency of any connector built atop that network. Ensuring connector properties may be possible by implementing them with Quality-of-Service (QoS) enabled middleware. Understanding how to better specify connector properties at the architectural level and how to derive constraints on those properties from network properties are important future research goals for us.

In the long term, we are interested in understanding how to integrate sytems built with distributed connectors into other parts of the software lifecycle. Understanding how to use a software architecture to deploy, manage, and evolve a distributed system is an extended research goal that we have begun to pursue through separate projects.

6. Conclusions

Ensuring interoperability is a critical issue in the quickly-emerging marketplace of heterogeneous software components. Using middleware as the only basis for interoperability in an application, however, is problematic. Middleware solutions allow system designers to overcome certain types of heterogeneity, but also impose architectural constraints on the components with which they interact. We believe that a better strategy is to design and describe connector properties *first*, during architectural design, then choose appropriate middleware solutions to build connectors that achieve those properties.

This paper has presented an approach that has the potential to make this strategy possible. The approach directly exploits architectural constructs (styles and connectors) and provides a principled, repeatable solution to the problem of bridging middleware. We have employed sets of both commercial (RMI, VisiBroker, E-Speak) and research (ILU, Polylith, Q) OTS technologies to test our hypothesis that software connectors are the proper mechanisms for supporting middleware-based implementation of architectures. Our results to date are quite promising: we were able to take these diverse middleware technologies and still build and use C2 connectors with the same basic integration techniques.

Any approach must begin with understanding the underlying properties, both shared and proprietary, of middleware technologies in order to make informed decisions about the best middleware for use in building a particular connector. Choosing appropriate middleware for a connector involves trading off many functional and non-functional properties. Middleware choice will be influenced by the properties of components which the connector must connect (language, method of communication), the properties of the platforms the connector must run on (taking into account things like resource constraints and performance), and non-functional properties like cost and vendor support.

With so many diverse middleware solutions available today, coupled with the complexity of modern component-based software applications, committing to a single middleware solution is a major risk. Encapsulating middleware in connectors mitigates this risk by allowing the middleware to be selected or changed without recoding individual components. The benefits of this work will accrue from better understanding of how to integrate the large amounts of legacy software at one's disposal and the knowledge of what (types of) components can be (re)used in an application and under what circumstances. In turn, in tandem with related academic and industry-led work, this research has the potential to influence the next generation of interoperability standards and provide the underpinning of a true, open component marketplace.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful and insightful comments that helped us to improve this paper for its final publication.

This material is partly based on work supported by the National Science Foundation under Grant No. CCR-9985441. This work is also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-00-2-0599, F30602-99-C-0174 and F30602-00-2-0615. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government. Work also sponsored in part by the U.S. Army Tank Automotive and Armaments Command and Xerox Corporation.

References

1. *Remote Method Invocation.* <http://java.sun.com:80/products/jdk/rmi/index.html>, Sun Microsystems, Inc.
2. *ILU–Inter-Language Unification.* <ftp://ftp.parc.xerox.com/pub/ilu/ilu.html>, Xerox Palo Alto Research Center.
3. *Real-time CORBA with TAO* (*The ACE ORB*). <http://www.cs.wustl.edu/ schmidt/TAO.html>
4. M. Abi-Antoun and N. Medvidovic, "Enabling the refinement of a software architecture into a design", in *Proc. 2nd Int. Conf. on The Unified Modeling Language* (*UML'99*), Fort Collins, CO, October, 1999.
5. J. Alateras, J. Mourikis, and T. Anderson, OpenJMS. Exolab, Inc., Melbourne, Australia, 2000–3. <http://openjms.sourceforge.net/>
6. R. Allen, and D. Garlan, "A formal basis for architectural connection", *ACM Trans. on Software Engineering and Methodology* **6**(3) (1997) 213–249.
7. Borland USA, *VisiBroker: Product Documentation.* <http://info.borland.com/techpubs/visibroker/>, Borland USA, 2003.
8. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and evaluation of a wide-area event notification service", *ACM Trans. on Computer Systems* **9**(3) (2001) 332–383.
9. G. Cugola, E. Di Nitto, and A. Fuggetta, "Exploiting an event-based infrastructure to develop complex distributed systems" in *Proc. 20th Int. Conf. on Software Engineering*, IEEE Computer Society. Kyoto, Japan, April, 1998, pp. 261–270.
10. E. M. Dashofy, N. Medvidovic, and R. N. Taylor, "Using off-the-shelf middleware to implement connectors in distributed software architectures", in *Proc. 21st Int. Conf. on Software Engineering* (*ICSE'99*), Los Angeles, CA, May 16–22, 1999, pp. 3–12.
11. E. Di Nitto and D. S. Rosenblum, "Exploiting ADLs to specify architectural styles induced by middleware infrastructures" in *Proceedings of the 21st International Conference on Software Engineering*, IEEE Computer Society, Los Angeles, CA, May, 1999, pp. 13–22.
12. W. Emmerich, "Software engineering and middleware: A roadmap", in *Proc. 22nd Int. Conf. on Software Engineering* (*ICSE 2000*)*: The Future of Software Engineering*, ed. A. Finkelstein, 2000, pp. 117–129.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional Computing Series, Addison-Wesley, Reading, MA, 1995.

14. Hewlett-Packard, E-Speak Homepage. <http://www.e-speak.net/>, 2001.
15. C. A. R. Hoare, "Communicating sequential processes", *Communications of the ACM* **21**(8) (1978) pp. 666–677.
16. P. Houston, *Building Distributed Applications with Message Queuing Middleware*, Microsoft Corporation, White Paper Report, 1998, p. 14. <http: //www.microsoft.com/ntserver/techresources/appserv/MSMQ/msmqdistributed.asp>
17. IBM Corporation, *WebSphere MQ — Product Overview.* <http://www-3.ibm.com/software/ts/mqseries/messaging/>, IBM, Website, 2003.
18. IBM Corporation, *Interface Tool for Java.* <http://www.alphaworks.ibm.com/tech/bridge2java>, Website, 2002.
19. Intrinsyc Software, Inc., *Highlighting Intrinsyc's Technologies: Intrinsyc J-Integra*, Report, 2001, p. 10. <http://www.intrinsyc.com/pdfs/whitepapers/whitepaper_jintegra.pdf>
20. JavaSoft., *Java Message Service API*, Sun Microsystems, Inc., Report Version 1.0.2b, August 27, 2001. <http://java.sun.com/products/jms/docs.html>
21. Lockheed Martin Corporation, *HARDPack Developer's Guide*, 91 pps., Lockheed Martin Corporation, Owego, NY, 1998.
22. D. C. Luckham and J. Vera, "An event-based architecture definition language", *IEEE Trans. on Software Engineering* **21**(9) (1995) 717–734.
23. M. J. Maybee, D. M. Heimbigner, and L. J. Osterweil, "Multilanguage interoperability in distributed systems", in *Proc. 18th Int. Conf. on Software Engineering*, IEEE Computer Society, Berlin, Germany, March, 1996, pp. 451–463.
24. N. Medvidovic, D. S. Rosenblum, and R. N. Taylor, "A language and environment for architecture-based software development and evolution", in *Proc. 21st Int. Conf. on Software Engineering* (*ICSE '99*), IEEE Computer Society, Los Angeles, CA, May, 1999, pp. 44–53.
25. N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages", *IEEE Trans. on Software Engineering* **26**(1) (2000) 70–93.
26. N. Medvidovic, "On the role of middleware in architecture-based software development", in *Proc. 14th Int. Conf. on Software Engineering and Knowledge Engineering* (*SEKE'02*), ACM, Ischia, Italy, July 15–19, 2002, pp. 299–306.
27. N. Medvidovic, N. R. Mehta, and M. Mikic-Rakic, "A family of software architecture implementation frameworks", in *Proc. 3rd IFIP Working Int. Conf. on Software Architectures*, Montreal, Canada, August, 2002.
28. N. Medvidovic, D. S. Rosenblum, D. Redmiles and J. Robbins, "Modeling software architectures in the unified modeling language", *ACM Trans. on Software Engineering and Methodology* **11**(1) (2002) 2–57.
29. M. Moriconi, X. Qian, and R. A. Riemenschneider, "Correct architecture refinement", *IEEE Trans. on Software Engineering* **21**(4) (1995) 356–372.
30. Object Management Group, ed., *The Common Object Request Broker: Architecture and Specification*, 946 pp., Object Management Group, 2001.
31. A. Oram, N. Minar and C. Shirky, *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, 1st edn. 432 pp., O'Reilly & Associates, 2001.
32. J. M. Purtilo, "The POLYLITH Software Bus", *ACM Transactions on Programming Languages and Systems* **16**(1) (1994) 151–174.
33. R. Sessions, *COM and DCOM: Microsoft's Vision for Distributed Objects*, John Wiley & Sons, New York, 1997.
34. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik,

"Abstractions for software architecture and tools to support them", *IEEE Trans. on Software Engineering* **21**(4) (1995) 314–335.
<http://citeseer.nj.nec.com/shaw95abstractions.html>

35. M. Shaw and D. Garlan, *Software Architecture: Perpectives on an Emerging Discipline*, 242 pp., Prentice Hall, 1996.
36. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, J. Robbins, K. A. Nies, P. Oreizy and D. Dubrow, "A component- and message-based architectural style for GUI software", *IEEE Trans. on Software Engineering* **22**(6) (1996) 390–406.
37. S. Williams and C. Kindel, *The Component Object Model: A Technical Overview.* <http://msdn.microsoft.com/library/techart/msdn_comppr.htm>, Microsoft Corporation, HTML, 1994.
38. J. A. Zinky, D. E. Bakken and R. E. Schantz, "Architectural support for quality of service for CORBA objects", *Theory and Practice of Object Systems*, April, 1997.
<http://www.dist-systems.bbn.com/papers/1997/TAPOS/>