Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures

Eric M. Dashofy
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
edashofy@ics.uci.edu

Nenad Medvidovic
Computer Science Dept.
University of Southern California
Los Angeles, CA 90089-0781
+1-213-740-5579
neno@usc.edu

Richard N. Taylor
Info. and Computer Science Dept.
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
+1-949-824-6429
taylor@ics.uci.edu

ABSTRACT
Software architectures promote development focused on modular building blocks and their interconnections. Since architecture-level components often contain complex functionality, it is reasonable to expect that their interactions will also be complex. Modeling and implementing software connectors thus becomes a key aspect of architecture-based development. Software interconnection and middleware technologies such as RMI, CORBA, ILU, and ActiveX provide a valuable service in building applications from components. The relation of such services to software connectors in the context of software architectures, however, is not well understood. To understand the tradeoffs among these technologies with respect to architectures, we have evaluated several off-the-shelf middleware technologies and identified key techniques for utilizing them in implementing software connectors. Our platform for investigation was C2, a component- and message-based architectural style. By encapsulating middleware functionality within software connectors, we have coupled C2's existing benefits such as component interchangeability, substrate independence and structural guidance with new capabilities of multi-lingual, multi-process and distributed application development in a manner that is transparent to architects.

Keywords
Connectors, middleware, software, architectures, C2.

1 INTRODUCTION
Software architectural styles, such as UNIX's pipe-and-filter style or blackboard architectures in artificial intelligence, are key design idioms [5, 18]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements (software components, connectors, etc.) and their overall interconnection structure. An issue associated with architectures that is magnified in comparison with conventional software design and programming is the existence of software connectors as top-level constructs.

In programming languages, connectors are primitive and implicit in, e.g., procedure calls and global variables. Since software components at the architectural level may contain complex functionality, it is reasonable to expect that their interactions will be complex as well. Modeling and implementing software connectors with potentially complex protocols thus becomes a key aspect of architecture-based development [1, 14, 23]. In architectures, connectors may, e.g., be separately compilable message routing devices, shared variables, table entries, buffers, instructions to a linker, dynamic data structures, procedure calls, initialization parameters, client-server protocols, pipes, SQL links between a database and an application, and so forth [3, 21].

While practitioners are typically intimately familiar with "connecting" software modules via, e.g., procedure calls, their understanding of other interconnection mechanisms, e.g., client-server protocols and message routers, is often minimal. Several commercial and research off-the-shelf (OTS) middleware software systems that explicitly implement such interconnection mechanisms are available: Field [20], SoftBench [2], Tooltalk [7], Q [9], Polylith [19], DCE [21], CORBA [16], ILU [26], COM/DCOM [22], and ActiveX [3]. Also, several object-oriented (OO) programming languages provide remote procedure call (RPC) mechanisms. A representative example is Java's Remote Method Invocation (RMI) system [24]. Unfortunately, the applicability of these mechanisms and tools to software architectures is not well understood. They are rarely used by architecture researchers in practice. With the exception of UniCon [23], the focus of researchers has instead generally been on formal modeling of connector protocols with implementation support for simple connections only.

Consequently, we have begun exploration of these issues and have used our work on the C2 architectural style [25] as a basis for using OTS middleware in the context of software architectures. Our goals were to understand the issues in adapting the different technologies and the tradeoffs among the levels of support they provide for the needs of software architectures. Our implementation infrastructure [11, 12] has enabled us to experiment with incorporating several existing middleware technologies into C2 architectures. We have built software connectors that use four middleware packages: the Q system, the Polylith software bus, Java's RMI facility, and ILU's distributed object system. In doing so, we developed and experimented with a set of techniques for integrating middleware into software connectors. The preliminary results of this work were reported in [10]. Our results to date suggest that our approach is general enough to be applicable across middleware technologies.

The remainder of this paper is organized as follows. Section 2 presents a brief overview of the support for connectors in current software architecture research. Section 3 summarizes the C2 architectural style and implementation infrastructure. This section also discusses an example C2-style architecture, which was used to demonstrate the integration of the middleware packages into C2 connectors. Sections 4 and 5 discuss our general approach to using middleware with software connectors and the results of doing so, respectively. Section 6 discusses our findings and the applicability of these findings to software architectures in general. A discussion of future work rounds out the paper.

2 OVERVIEW OF THE ROLE OF CONNECTORS IN SOFTWARE ARCHIECTURES

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 [4]. 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 [1], UniCon [23], and Rapide [8].

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) [6]. 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 Section 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.

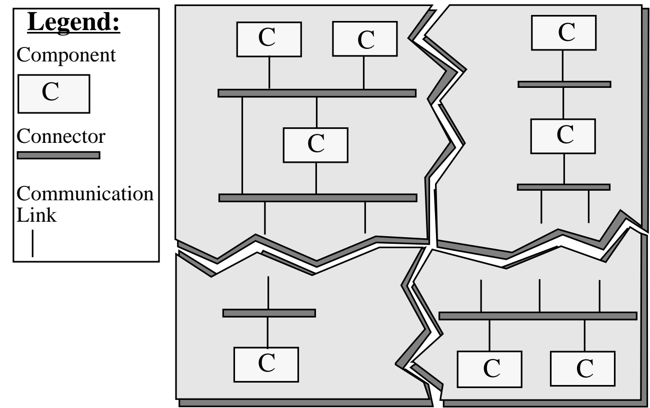


Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

3 OVERVIEW OF THE C2 ARCHITECTURAL STYLE

We chose the C2 architectural style as a foundation upon which to explore issues of integrating middleware with software connectors. The C2 style has an explicit notion of connectors as first-class entities and provides facilities to explore specific properties of software connectors such as filtering, routing, and broadcasting (described in more detail below). Further, the style is well-suited to a distributed environment, allowing us to leverage the networking capabilities of middleware technologies. The style supports a paradigm for composing systems in which components may be running in a distributed, heterogeneous environment without shared address spaces, architectures may be changed dynamically, multiple users may be interacting with the system, multiple user interface toolkits may be employed, multiple dialogs may be active (and described in different formalisms), and multiple media types may be involved.

For those unfamiliar with the C2 style, it is described in [25]. The C2 style can be informally summarized as a network of concurrent components hooked together by connectors, i.e. message routing devices. Components and connectors both have a defined top and bottom. The top of a component may be connected to the bottom of a single connector and the bottom of a component may be connected to the top of a single connector. There is no bound on the number of components or connectors that may be attached to a single connector (see Fig. 1). All communication in a C2 architecture is solely achieved by exchanging messages. Message-based communication is extensively used in distributed environments for which C2 is suited.

Each component may have its own thread(s) of control. This simplifies modeling and implementation of multi-component, multi-user, and concurrent applications and enables exploitation of distributed platforms. Note that separating components into different threads of control is not a requirement. Moreover, a proposed architecture is distinct from its implementation(s) so that it is indeed possible for

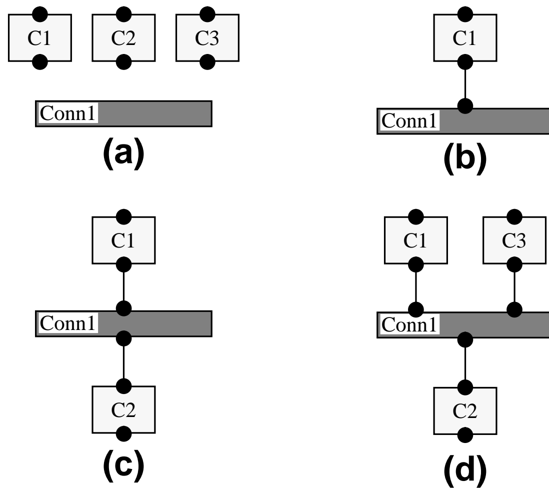


Fig. 2. C2 connectors have *context reflective interfaces*. Each C2 connector is capable of supporting any number of C2 components.
(a) Software architect selects a set of components and a connector from a design palette. The connector has no communication ports, since no components are attached to it.
(b-d) As components are attached to the connector to form an architecture, the connector creates new communication ports to support component intercommunication.

components to share threads of control. This separation of architecture from implementation is a key aspect of our approach to integrating middleware technologies into C2, as discussed in Section 4.

Finally, there is no assumption of a shared address space among C2 components. Any premise of a shared address space could be unreasonable in an architectural style that allows composition of heterogeneous, highly distributed components, developed in different languages, with their own threads of control, internal structures, and domains of discourse.

3.1 C2 Connectors

Connectors bind components together into a C2 architecture. They may be connected to any number of components as well as other connectors. A connector's primary responsibility is the routing and broadcast of messages. A secondary responsibility is message filtering.

Connectors may provide a number of filtering and broadcast policies for messages, such as the following:

- *no filtering* — each message is sent to all connected components on the given side of the connector (top or bottom).
- *notification filtering* — each notification is sent to only those components that have registered for it.
- *prioritized* — the connector defines a priority ranking over its connected components, based on a set of evaluation criteria specified by the software architect during the construction of the architecture. This connector then sends a notification to each component in order or priority until a termination condition has been met.
- *message sink* — the connector ignores each message sent to it. This is useful for isolating subsystems of an architecture as well as incrementally adding components to an existing architecture. A developer can connect a new component to the architecture and then "turn on" its connector, by changing its filtering policy, when the component is ready to start sending and receiving messages.

A unique feature of C2 connectors is that they have context reflective interfaces: a C2 connector is not defined to have a particular interface; instead, its interface is a function of the interfaces of components attached to it (see Fig. 2). A given C2 connector must be capable of supporting (message-based) communication among any C2 components. We have exploited this feature of C2 connectors to support both specification-time and run-time evolution of C2-style architectures [14, 15].

C2 connectors and components are joined with intermediary entities called "ports." Ports form message pathways between connectors and components, but they do not play an active role in filtering messages.

3.2 Implementing C2 Style Architectures

The ultimate goal of any software design and modeling endeavor is to produce the executable system. An elegant and effective architectural model is of limited value unless it can be converted into a running application. Doing so manually may result in many problems of consistency and traceability between an architecture and its implementation. For example, it may be difficult to guarantee or demonstrate that a given system correctly implements an architecture. Furthermore, even if this is currently the case, one has no means of ensuring that future changes to the system are appropriately traced back to the architecture and vice-versa. It is, therefore, desirable, if not imperative, for architecture-based software development approaches to provide source code generation tools.

To support implementation of C2 architectures, we developed an extensible framework of abstract classes for C2 concepts such as components, connectors and messages, as shown in Fig. 3. This framework is the basis of development and middleware integration in C2. As we will discuss, the framework encapsulates all access to integrated middleware, ensuring that the use of middleware is transparent to an architect, and, indeed, to the implementor of a particular architecture. In Section 5, we will show that middleware can be further encapsulated completely within the connector elements of the framework. The framework implements interconnection and message passing protocols. Components and connectors used in C2 applications are subclassed from the appropriate abstract classes in the framework. This guarantees their interoperability, eliminates many repetitive programming tasks, and allows developers of C2 applications to focus on application-level issues. The framework supports a variety of implementation configurations for a given architecture: the entire resulting system may execute in a single thread of control, or each component may run in its own thread of control or operating system (OS) process.

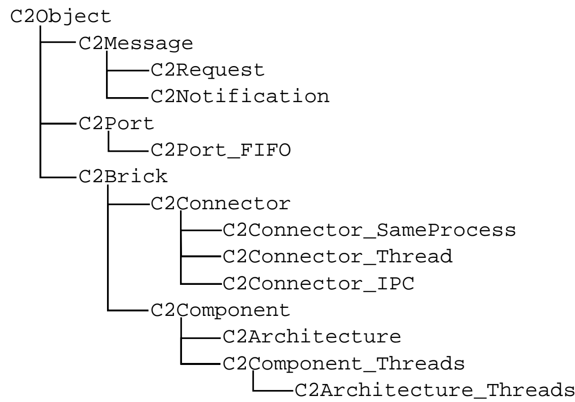


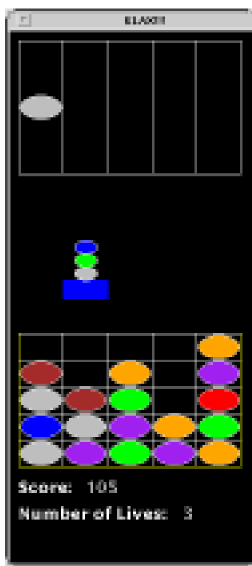
Fig. 3. C2 object-oriented class framework.

3.3 An Example C2 Application

The example application that was used in our investigation of OTS middleware integration in C2 is a version of the video game KLAX. A description of the game is given in Fig. 4. This particular application was chosen because game play imposes some real-time constraints on the application, bringing performance issues to the forefront.

The architecture of the system 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 which encapsulate the game's state. The game state components respond to requests and emit notifications of internal state changes. Notifications are directed to the next level where they are received by 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 game state change notifications to determine the state of the game in progress.



KLAX Chute
Tiles of random colors drop at random times and locations.

KLAX Palette
Palette catches tiles coming down the Chute and drops them into the Well.

KLAX Well
Horizontal, vertical, and diagonal sets of three or more consecutive tiles of the same color are removed and any tiles above them collapse down to fill in the newly-created empty spaces.

KLAX Status

Fig. 4. A screenshot and description of our implementation of the KLAX video game.

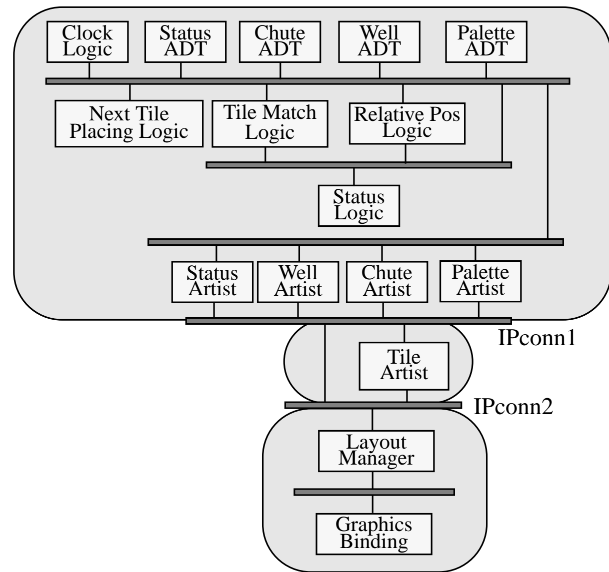


Fig. 5. Conceptual C2 architecture for KLAX. Shaded ovals represent process boundaries in the three-process implementations of KLAX.

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.

The *GraphicsBinding* component 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 into requests to the artist components.

The KLAX application was used as a testbed for our research on middleware. We used the partitioning shown in Fig. 5 for testing OTS middleware technologies, although other partitionings are possible. Two KLAX implementations were built using the C++ and Java frameworks shown in Fig. 3. Both implementations consist of approximately 8000 lines of commented code, in addition to the base framework's 3000 lines of code. A variation of the architecture shown in Fig. 5 was also used as the basis of a distributed, multi-player KLAX application implemented using the Java framework. In this variation each player executes a copy of KLAX on his own machine. A player competes against other game participants by issuing requests to a central *GameServer* to, e.g., add an extra tile to a given player's chute. 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

4 THE ROLE OF MIDDLEWARE

Middleware is a potentially useful tool when building software connectors. First, it can be used to bridge thread, process and network boundaries. Second, it can provide pre-built protocols for exchanging data among software components or connectors. Finally, some middleware packages include features of software connectors such as filtering, routing, and broadcast of messages or other data.

4.1 Middleware Evaluation Criteria

When evaluating OTS middleware technologies, we focused on several factors. We do not expect a single technology to satisfy all of these requirements. The selection process must be at least partially based on the characteristics and needs of a specific application:

- *inter- and intra-process communication support* — a distributed application is likely to contain a mix of components that execute in a *single thread* of control, in *different threads* of control (but in the same process), and in *different processes*, some of which will reside on different machines. If a given middleware technology effectively supports only interprocess communication, its utility is limited and additional types of middleware may need to be employed. Note that multiple types of middleware in an application may indeed be preferable, as each may optimize a particular type of communication.
- *features of software connectors* — a middleware technology may only provide the ability for two processes to exchange data. The needs of software connectors are broader: event routing (e.g., broadcast, multicast, point-to-point), filtering, registration, and so forth [17]. If such features are not supported, additional infrastructure must be provided before such a technology may be used in a distributed architecture, such as the one depicted in Fig. 5.
- *platform and language support* — software architectures, and C2 architectures in particular, are intended to support the development of distributed systems, built out of components which are potentially implemented in different programming languages and executing on multiple platforms. An interconnection technology that supports multilingual and multi-platform applications is thus a better candidate for integration than one that does not. The penalties (e.g., adoption costs, performance) accrued by using a technology that only supports a single language and/or platform may outweigh any benefits of using it.
- *communication method* — similarly to the different types of connectors at the architectural level (see Section 1), methods of communication across middleware technologies vary and can include remote procedure calls (RPC), message passing, passing object references, shared memory, and so forth. A middleware technology that is not suited to an architectural style may cause implementation difficulties when used in the context of that style. However, as we will show, it is possible to implement connectors that make translations from one communication method to another (e.g. RPC to message-passing) to fit within a given architectural style.
- *ease of integration and use* — if integrating an OTS technology into the implementation infrastructure and/or its use in an application requires a substantial amount of effort, its effectiveness and power may be rendered irrelevant. For example, if an interconnection tool assumes that it is the application's main thread of control, it is not well suited for use with C2, since C2 mandates that all components execute independently of each other. The amount of work required to integrate Q, Polylith, RMI and ILU (see Section 5) was relatively minor, typically only requiring additions of message routing or marshaling code.
- *multiple instances in an application* — one benefit of distributed systems is that they do not have to depend on a single set of resources, thus avoiding performance bottlenecks. Analogously, it may be useful to physically distribute the very tool used to interconnect a distributed system. Centralized OTS middleware tools that use a single point of communication form potential bottlenecks and single points of failure.
- *support for dynamic change* — for an important class of safety- and mission-critical software systems, such as air traffic control or telephone switching systems, shutting down and restarting the system for upgrades incurs unacceptable delays, increased cost, and risk. Support for runtime modification is thus a key aspect of these systems. A middleware technology that does not support dynamic change is not an adequate candidate for them.
- *performance* — performance is a key issue in systems with real-time requirements. For example, in the KLAX application from Section 3, several hundred messages may be generated every second. The ability to efficiently ferry these messages among the components and across process boundaries is paramount.

4.2 Supporting Cross-Process Communication

Of all the issues noted above, the need for effectively supporting inter-process communication is particularly important to us. Distributed applications require this and most middleware technologies emphasize this capability. Consequently we focused particular attention on how middleware could be used to support connectors transparently spanning process boundaries. After examining the field of available middleware packages and their capabilities, we examined four possible approaches to connecting parts of a C2 application running in multiple processes and possibly on multiple machines. These approaches are independent of the choice of underlying middleware; they do not depend on the properties of any particular middleware package. Two of these approaches divide an architecture along a single communication port, while the other two do so along a connector, as indicated in Fig. 5.

4.2.1 Linking Ports across Process Boundaries

The first approach that we examined and attempted to implement involved linking two C2 ports across a process or a machine boundary using a middleware package to bridge those boundaries. All accesses to the middleware technology would be entirely encapsulated within the port entity and would not be visible to architects or developers. The single-process implementation of a C2 connector links two ports together by having each port contain a reference to the other one. In this way, the ports can call methods on each other, sending

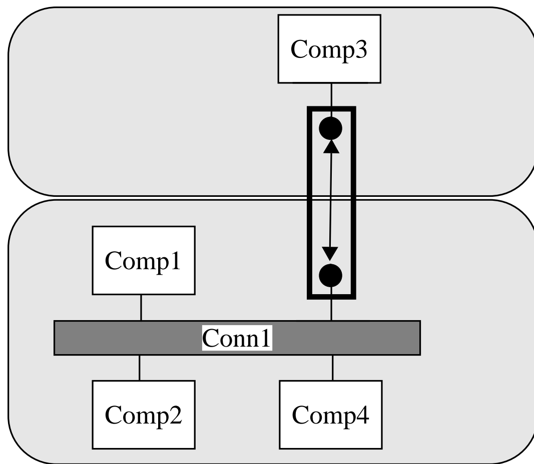


Fig. 6. Two communication ports in separate processes comprise a single "virtual port." For clarity, we do not highlight component and connector ports. Shaded ovals represent process boundaries.

messages as method parameters. Our intent was to simply use the middleware to exchange port references across process boundaries and use the existing technique for message passing.

We attempted to implement this strategy, but found that it was infeasible for several reasons. Most importantly, ports, being complex objects, were not easily serializable. For each of the middleware technologies we evaluated, any objects sent across a process or network boundary must first be serialized into a byte stream. C2 ports contain references to complex C2 objects to which they are attached (connectors, components, and entire architectures), which would, in turn, also have to be serialized — hardly a reasonable approach. Secondly, references to objects are typically not preserved across process boundaries since all network data is passed by copy instead of by reference. Thus, even if we could overcome the serialization issue and pass port objects through the network, a connection made by using the references to them would not be preserved over the network. This experience indicated that any passing of complex objects across a process or network boundary would be impossible using our available middleware technology.

With this knowledge, we refined our approach: rather than attempting to send whole C2 port objects across process and network boundaries, we simply sent messages, which consist only of data and are easily marshaled. In this approach, two ports are created, one per process, to simulate a single "virtual port," as shown in Fig. 6. Rather than sending a reference to itself to the other port, each port simply sends messages.

4.2.2 Linking Connectors across Process Boundaries

Sharing communication ports across process boundaries gave us fine-grained control over implementing an architecture as a multi-process application. However, it required additional functionality in the C2 implementation framework and did not isolate the change to the appropriate abstraction: the connector. In order to remedy this, we devised two connector-based approaches. Both of these approaches consist of implementing a single conceptual software connector using two or more actual connectors that are linked across process or network boundaries. Each actual connector thus becomes a segment of

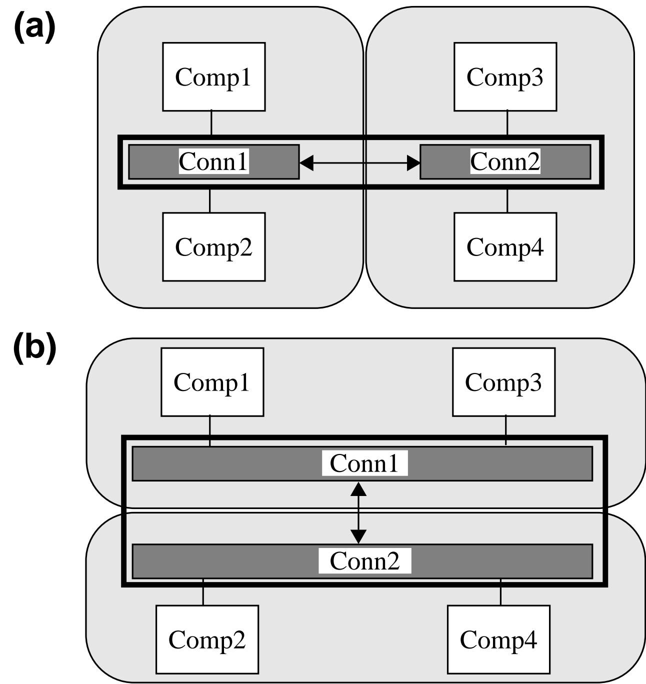


Fig. 7. Connectors as a primary vehicle for interprocess communication. A single conceptual connector can be "broken up" vertically (a) or horizontally (b) for this purpose. Shaded ovals represent process boundaries.

a single "virtual connector." All access to the underlying middleware technology is encapsulated entirely within the abstraction of a connector, meaning that it is unseen by both architects and developers.

We call the first approach "lateral welding," depicted in Fig. 7(a). Messages sent to any segment of the multi-process connector are broadcast to all other segments. Upon receiving a message, each segment has the responsibility of filtering and forwarding it to components in its process as appropriate. Only messages are sent across process boundaries.

While the lateral welding approach allowed us to "vertically slice" a C2 application, we also developed an approach to "horizontally slice" an application, as shown in Fig. 7(b). This approach is similar to the idea of lateral welding: a conceptual connector is broken up into top and bottom segments, each of which exhibits the same properties as a single-process connector to the components attached above and below it, respectively. However, the segments themselves are joined using the appropriate middleware.

When used with a middleware technology that supports dynamic change at run-time, all of these approaches, both using ports and connectors, can be used to build applications where processes can join and leave a running application. Only a small bit of additional infrastructure is required to notify a running architecture that a new process is joining or leaving the application.

5 USE OF OTS MIDDLEWARE TECHNOLOGIES

To explore the use of OTS middleware with software connectors, we chose four representative technologies from the field of available middleware packages. These were Q, an

RPC system, Polylith, a message bus, RMI, a connection mechanism for Java objects, and ILU, a distributed objects package.

5.1 Q

The Q system [9], developed at the University of Colorado, is intended to provide interoperability support for multilingual, heterogeneous component-based systems. Q presents a layer of functionality between software components communicating across process boundaries. It is based on remote procedure calls (RPC) and provides support for marshaling and unmarshaling of arbitrarily complex type structures. Q also supports placement of components executing in a single thread or in multiple threads of control inside a single process. It ensures the proper communication of multi-threaded components with other parts of a system. Q addresses the issue of performance by adding an asynchronous message interface on top of a standard RPC interface, so that processor time is used for interprocess communication only when it is known that data is pending.

Q uses a remote procedure call (RPC) mechanism for communication, which is dissimilar to C2's message-based style. Nonetheless, we easily emulated message passing using RPC by passing serialized messages as parameters in remote calls. Q supports systems built in several languages: C/C++, Ada, Java, Tcl, Lisp, and Prolog. It was originally built for the UNIX platform, although its Java interface presents the potential for moving to other platforms. We have made use of its support for C/C++ and Ada with the intent to exploit its support for Java in the near future.

Our approach to integrating Q with the C2 implementation infrastructure consisted of encapsulating Q inside a C2 connector (we refer to it as a "Q-C2 connector" below). Q is not a software bus, so it does not support typical connector-like features, such as event registration, filtering, and routing. However, this layer of support is added easily in a Q-C2 connector.

A Q-C2 connector exports the same interface as a regular C2 connector, so architects attach components to it in the usual manner. Internally, however, a Q-C2 connector provides a mechanism for communicating across process boundaries via Q. At each process boundary, a conceptual C2 connector is "broken up" into two or more Q-C2 connectors, one per process, as shown in Fig. 7b. When using Q-C2 connectors, all processes containing C2 subarchitectures must register with a single "name server." All links across process boundaries are specified in the Q-C2 connector, by naming the attached connectors, and are maintained by Q at execution time. Clearly, care must be taken to ensure that there are no naming conflicts, i.e., that multiple Q-C2 connectors do not share a name.

Given that we can explicitly specify the connections among Q-C2 connectors in an architecture, a single instance of Q is sufficient to support the needs of an architecture. Since Q is UNIX-based, it supports addition and removal of processes at execution time. Any additional support for dynamism, such as transactions, state preservation during change, or component (i.e., process) replacement, must be built on top of Q.

We used Q to generate a multi-process version of KLAX, shown in Fig. 5. Connectors *IPconn1* and *IPconn2* were used at process boundaries. The rest of the application remained identical to single-process KLAX. This three-process configuration allowed us to explore issues in supporting multilingual applications in C2. For example, we were able to replace the "middle" process in KLAX, where the *TileArtist* component and both connectors were initially implemented in C++, with their Ada implementations. This can be done at specification or execution time. If the change is made at run-time, a part of the game state is lost, as no one receives the notifications issued by components in the "top" process or requests issued by the "bottom" process components during the course of the change. The performance of this variation of KLAX easily exceeded human reaction time if the *ClockLogic* component used short time intervals.

5.2 Polylith

The Polylith software bus was developed at the University of Maryland [19]. Polylith was built to allow several parts of an application to communicate across process boundaries using messages made up of arbitrarily complex type structures. Polylith uses messages for communication, which made it well-suited for implementing C2-style connectors. Polylith can transfer messages among processes running on a single machine or on multiple machines using the TCP/IP networking protocol. The Polylith toolkit is implemented in C and runs on several variants of UNIX. Polylith supports applications developed in C/C++; support for additional programming languages is under development.

Polylith is inherently built to communicate among UNIX processes. Although there is no support for multithreading in Polylith, multiple threads within a process are allowed in principle. Polylith has support for marshaling and unmarshaling of C basic types and structures. The Polylith bus itself runs in its own process and acts as a message queue for other processes, which are individually responsible for periodically sending and retrieving messages to and from the bus.

Like the Q-C2 connector, the "Polylith-C2" connector is an extension of the standard, in-process C2 connector. All access to Polylith is done within the C2 connector, and is transparent. Components can attach themselves to a Polylith-C2 connector in the usual manner.

The process-level structure of a C2 application that uses Polylith is defined statically, i.e., at compile time, using a proprietary language called MIL. The MIL code can be generated automatically in a fairly straightforward manner. As a software bus, Polylith has the ability to route messages at the process level, but it was necessary for us to implement our own intra-process routing mechanisms. There is no support for message filtering in Polylith.

The current Polylith toolkit uses the UNIX process scheduler for all process scheduling. Polylith applications with specific scheduling needs must explicitly make system-level calls from within the application. Such performance limitations became problematic when Polylith-C2 was used in the implementation of the KLAX application from Section 3. The implementation suffered from poor performance due to the UNIX process scheduler giving large time slices to each process, resulting in

messages being handled in bursts rather than in a fluid manner. This may be unacceptable in a real-time application such as KLAX. The authors of Polylith acknowledge this problem; an experimental, as yet unreleased version of Polylith alleviates this shortcoming.

5.3 RMI

Java's Remote Method Invocation (RMI) [24] is a technology developed by Sun Microsystems to allow Java objects to invoke methods of other objects across process and machine boundaries. RMI supports several standard distributed application concepts, namely registration, remote method calls, and distributed objects. Currently, RMI only supports Java applications, but there is indication of a forthcoming link between RMI and CORBA that would remedy this.

Each RMI object that is to be shared in an application defines a public interface (a set of methods) that can be called remotely. This is similar to the RPC mechanism of Q. These methods are the only means of communication across a process boundary via RMI. Because RMI is not a software bus, it has no concept of routing, filtering, or messages. However, Java's built-in serialization and deserialization capabilities handle marshaling of basic and moderately complex Java objects, including C2 messages.

RMI is fully compatible with the multithreading capabilities built into the Java language, and is therefore well suited for a multithreaded application. It allows communication among objects running in different processes which may be on different machines. Communication occurs exclusively over the TCP/IP networking protocol.

Like the Polylith- and Q-integrated connectors, the RMI-C2 connector we developed has all the capabilities of a single-process C2 connector. Additionally, it has the ability to register and deregister itself at run-time with the Java-RMI name server, and to be linked to other registered connectors. All access to RMI facilities is encapsulated within the connector and is transparent.

Minimal modification was required to convert the existing C2 KLAX application into a multi-process application that uses RMI-C2 connectors. RMI supports application modification at run-time, a capability enabled by Java's dynamic class loading. The performance of the three-process implementation of KLAX using RMI-C2 was satisfactory. Another variation of the KLAX application built using RMI-C2 connectors was multiplayer KLAX. This variation allowed players to remotely join a game already in progress and compete against other participants.

RMI's properties make it ideal for use within a Java C2 application. Its native support in Java 1.1 makes it more available to architecture implementors than third party alternatives. Also, using software connectors that work with RMI does not preclude an application implemented partially or completely in Java from using another middleware technology, such as Q or ILU, as well.

5.4 ILU

Xerox PARC's ILU (Inter-Language Unification) [26] was developed as a free CORBA-like object brokering system. Functionally, it is similar to Java RMI, allowing objects to call methods on other objects across process or network boundaries. ILU is different from RMI in that it has wide platform and language support: C, C++, Java, Python, LISP, Modula-3, Perl and Scheme on both Windows and UNIX platforms. The current ILU implementation can be thought of as a CORBA Object Request Broker (ORB), but ILU is not yet fully CORBA compliant.

Like RMI, each ILU object that is to be shared in an application defines a public set of methods that can be called remotely. There is no inherent concept of messages in ILU, but messages can be passed as parameters in remote method calls. Similarly to Q, ILU has the ability to serialize moderately complex objects across language boundaries. As with other distributed object systems, references are not preserved across the serialization boundary. ILU does not include a name server, but it facilitates object registration through a method called "simple binding" that is part of the ILU package. Our integration of ILU with C2 was done using the Java implementations of the C2 framework and the ILU package. The ILU-C2 connector thus created has all the capabilities of an in-process C2 connector, but it is also capable of lateral connection to ILU-C2 connectors in other processes. Again, all access to ILU is done entirely within the connector, in a manner that is transparent to architects and developers.

ILU takes full advantage of Java's multithreading capabilities and works in multithreaded applications written in other languages, even if such threading is provided by the operating system rather than the language itself. This makes it well suited for real-time, asynchronous message passing architectures, such as C2-style architectures. Minimal modification was required when converting a single-process C2 application to a multi-process C2 application. ILU allows objects to be registered and deregistered at run-time, therefore enabling dynamic application construction at run-time. We utilized this feature to demonstrate a set of components and connectors joining a larger, already executing application at run-time.

5.5 Simultaneous Use of Multiple Middleware Technologies

Each middleware technology we evaluated has unique benefits. By combining multiple such technologies in a single application, 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.

The advantages of combining multiple middleware technologies within software connectors are manifold. In the absence of a single panacea solution that supports all required platforms, languages, and network protocols, the ability to leverage the capabilities of several different middleware technologies significantly widens the range of applications that can be implemented within an architectural style such as C2.

We combined our implementations of ILU-C2 and RMI-C2 connectors in a version of the KLAX application. We were able to do so with no modification to the C2 framework or the connectors themselves by combining the lateral welding technique shown in Fig. 7(a) with the horizontal slicing

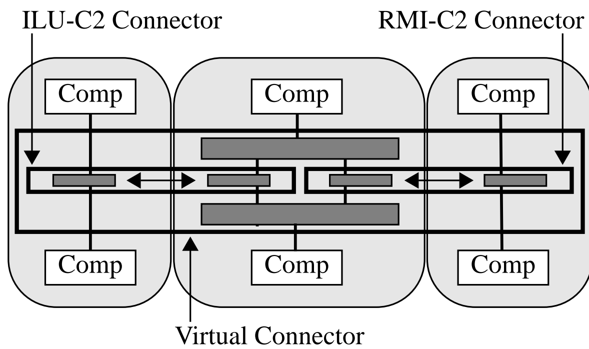


Fig. 8. An example of a three-process C2 application using different OTS middleware types. A single virtual connector is implemented with two in-process and two multi-process connectors. The in-process connectors facilitate message passing between the multi-process connectors. Shaded ovals represent process boundaries.

technique shown in Fig. 7(b). An example of this combined binding method is shown in Fig. 8. The approach shown in this figure creates a three-process "virtual connector" using two in-process C2 connectors to laterally bind two multi-process connectors. This approach works for any combination of OTS connectors that use the lateral welding technique. An alternative approach would have been to create a single connector that supported both ILU and RMI, but this would have required changes to the framework. Using the technique shown Fig. 8 avoids this difficulty with a slight efficiency cost due to the addition of in-process connectors to bind the multi-process connectors.

6 DISCUSSION

Because software connectors provide a uniform interface to other connectors and components within an architecture, architects need not be concerned with the properties of different middleware technologies as long as the technology can be encapsulated within a software connector. Internally, however, connectors based on different middleware technologies have different abilities. Implementors of a given architecture can use this knowledge to determine which middleware solutions are appropriate in a given implementation of an architecture. In this way, encapsulating middleware functionality within software connectors maintains the integrity of an architectural style by keeping it separate from implementation-dependent factors such as how to bridge process boundaries within a single architecture.

Currently, a major challenge in computing is the integration of existing legacy systems with new software capabilities. Another aspect of this problem is retrofitting existing components' interfaces to use them in new contexts. By encapsulating legacy systems and new software in software component wrappers as described in [11, 12] and binding these components together with middleware-integrated software connectors, building new software systems with legacy packages becomes less difficult. Consider the case where a legacy server must be integrated with new client software. If the server and client packages are both encapsulated within software components, middleware-enabled connectors such as those described above can be used to bridge language, platform, and network boundaries. The middleware transparently performs platform- and language-independent data transfer across the network, allowing the old and new components to communicate in a manner that is defined by the architecture rather than the implementation.

Software connectors have been embraced as a critical abstraction by software architecture researchers. Connectors remove from components the responsibility of knowing how they are interconnected. They also introduce a layer of indirection between components. The potential penalties paid due to this indirection (e.g., performance) should be outweighed by other benefits of connectors, such as their encapsulation of complex intercommunication protocols that can be reused relatively inexpensively across applications. Modeling and implementation of software connectors with potentially complex protocols thus becomes an important aspect of architecture-based development.

Our research to date has identified advantages and shortcomings in several middleware packages when used in a real-time component- and message-based architectural style. Using techniques such as those described here, we speculate that such middleware could be integrated with other architectural styles.

7 FUTURE WORK

In the process of integrating four different OTS middleware technologies with the C2 implementation infrastructure, we developed several techniques for using middleware that show potential for general applicability. In our ongoing project we plan to widen our base of integrated middleware technologies to include technologies such as CORBA and DCOM [22]. Based on our experience, we believe that integrating other middleware will proceed similarly to that described here. We intend to further refine our existing connectors to improve performance and capabilities. We also intend to analyze the efficiency implications of using OTS middleware. Lastly, our work suggests that the techniques we have developed can be used to integrate OTS middleware within other architectural styles that have explicit notions of connectors. We intend to explore this idea.

8 ACKNOWLEDGEMENTS

The authors would like to thank Peyman Oreizy and Ken Anderson for their work on various aspects of the C2 style and, in particular, for their efforts in integrating Q with the C2 implementation infrastructure.

This effort is partially sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. Approved for public release — distribution unlimited. 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.

This effort is also sponsored by Hughes Electronics Corporation under UC MICRO grant number 97-177.

REFERENCES

1. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, July 1997.
2. M. R. Cagan. The HP SoftBench Environment: An Architecture for a New Generation of Software Tools. *Hewlett-Packard Journal*, June 1990.
3. D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, 1996.
4. D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.
5. D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
7. A. Julienne and B. Holtz. *Tooltalk and Open Protocols: Inter-Application Communication*. SunSoft Press/Prentice Hall, April 1993.
8. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.
9. M. J. Maybee, D. H. Heimbigner, and L. J. Osterweil. Multilanguage Interoperability in Distributed Systems: Experience Report. In *Proceedings of the Eighteenth International Conference on Software Engineering*, Berlin, Germany, March 1996.
10. N. Medvidovic, E. Dashofy, and R. N. Taylor. Employing Off-the-Shelf Connector Technologies in C2-Style Architectures. *Proceedings of the California Software Symposium*, October 1998.
11. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)* and *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997.
12. N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, October-December 1997.
13. N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland, September 1997.
14. N. Medvidovic, R. N. Taylor, and D. S. Rosenblum. An Architecture-Based Approach to Software Evolution. In *Proceedings of the International Workshop on the Principles of Software Evolution*, Kyoto, Japan, April 20-21, 1998.
15. P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, April 1998, Kyoto, Japan.
16. R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1996.
17. D.E. Perry. Software Architecture and its Relevance to Software Engineering. *Coord'97*, September 1997.
18. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.
19. J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, January 1994.
20. S. P. Reiss. Connecting Tools Using Message Passing in the Field Environ*ment. IEEE Software, pages 57–66, July 1990.*
21. A. Schill, editor. *DCE — The OSF Distributed Computing Environment*. Proceedings of the International DCE Workshop, Karlsruhe, Germany, Springer Verlag, October 1993.
22. R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, New York, NY, 1997.
23. 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 Transactions on Software Engineering*, April 1995.
24. Sun Microsystems, Inc. Remote Method Invocation. http://java.sun.com:80/products/jdk/rmi/index.html
25. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, June 1996.
26. Xerox Palo Alto Research Center. ILU — Inter-Language Unification. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html