Employing Off-the-Shelf Connector Technologies in C2-Style Architectures

Nenad Medvidovic neno@ics.uci.edu
Eric M. Dashofy emdashof@uci.edu
Richard N. Taylor taylor@ics.uci.edu

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.

ABSTRACT

Software architectures shift the focus of developers from lines of code to coarser-grained building blocks and their overall interconnection structure. 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 thus becomes a key aspect of architecture-based development. While several software connector technologies are available, their utility in the context of emerging software architectures is generally not well understood and they are hence rarely used by architecture researchers and practitioners. A goal of our study is understanding the tradeoffs among these technologies. In this paper we describe a set of exercises in which off-the-shelf connector and middleware technologies were integrated for use with C2-style architectures. Several possible integration techniques were devised and evaluated in the process. While further trials are warranted, initial results suggest that our approach is applicable across connector technologies.[1]

1 INTRODUCTION

Software architectural styles, such as Unix's pipe-and-filter style or blackboard architectures in artificial intelligence, are key design idioms [4, 16]. 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 implementation of software connectors with potentially complex protocols thus becomes a key aspect of architecture-based development [1, 11, 21]. 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) software systems that explicitly implement such interconnection mechanisms are available: Field [18], SoftBench [2], Tooltalk [6], Q [8], Polylith [17], DCE [19], CORBA [15], COM/DCOM [20], and ActiveX [13]. Also, several object-oriented (OO) programming languages provide remote procedure call (RPC) mechanisms. A representative example is Java's Remote Method Invocation (RMI) system [22]. Unfortunately, the applicability of these mechanisms and tools to software architectures is not well understood. They are rarely used by architecture researchers in practice. Instead, the focus has generally been on formal modeling of connector protocols with implementation support for simple connections only.

We have used our work on the C2 architectural style [23] as a basis for using OTS connector technologies 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. In particular, our implementation infrastructure [9, 10] has enabled us to experiment with incorporating several existing connector technologies into C2 architectures. This paper describes our results to date. We have been able to successfully expand our implementation infrastructure to support connectors based on the Q system, the Polylith software bus, and Java's RMI facility. In the process, we have developed and experimented with a set of possible integration techniques. Although we are currently still actively investigating this subject, our initial results suggest that our approach is general enough to be applicable across OTS connector technologies.

The remainder of this paper is organized as follows. Section 2 presents an 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 integrations of the OTS connector technologies. Sections 4 and 5 discuss our general

1. Effort 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.

approach and the results of OTS connector integrations, respectively. A conclusion and discussion of future work round out the paper.

2 OVERVIEW OF THE ROLE OF CONNECTORS IN SOFTWARE ARCHITECTURES

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 [3]. 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 [21], and Rapide [7].

Wright is an architecture description language (ADL) whose particular focus is on formally specifying protocols of interaction among components in an architecture. To this end, it employs a subset of communicating sequential processes (CSP) [5]. 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 (correct) implementation of connectors.

UniCon, on the other hand, focuses on implementing connectors. To that end, it predefines a set of supported connector types: pipe, file I/O, procedure call(s), data access(es), and remote procedure call(s). UniCon's shortcoming is that it supports a small set of simple connectors for which implementation is either already provided by the chosen underlying programming language or 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 connector technologies discussed in Section 1.

Finally, 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 for each connector *type*.

There is, therefore, a need for an approach where powerful and extensible connector modeling formalisms (Wright) are coupled with connector implementation support (UniCon) and architecture simulation and code generation (Rapide). This is a complex task. Our hypothesis is that this task can be made easier by building upon existing interconnection mechanisms.

3 OVERVIEW OF THE C2 ARCHITECTURAL STYLE

We use the C2 architectural style as a platform for investigating the issues in integrating OTS connector

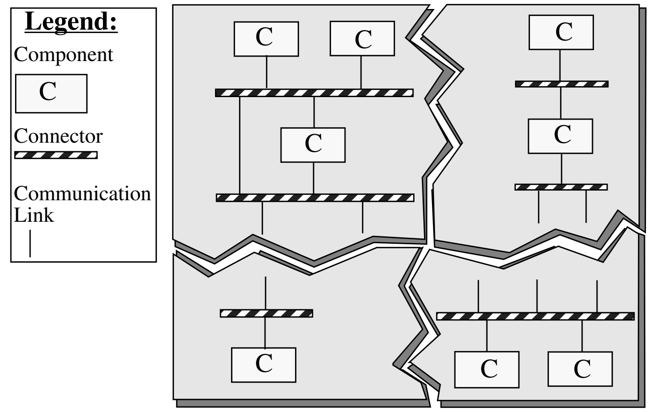

Fig. 1. A sample C2 architecture. Jagged lines represent the parts of the architecture not shown.

technologies. C2 is designed to support the particular needs of applications that have a graphical user interface aspect, with the potential for broader applicability. The style supports a paradigm for composing systems in which: components may be written in different programming languages, 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 toolkits may be employed, multiple dialogs may be active (and described in different formalisms), and multiple media types may be involved.

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 has a top and bottom domain. The top domain specifies the set of notifications to which a component responds, and the set of requests that the component emits up an architecture. The bottom domain specifies the set of notifications that this component emits down an architecture and the set of requests to which it responds. Central to the architectural style is a principle of limited visibility or *substrate independence*: a component within the hierarchy can only be aware of components "above" it and completely unaware of components which reside "beneath" it.

Each component may have its own thread(s) of control, a property also suggested by the asynchronous nature of tasks in the GUI domain. It simplifies modeling and programming of multi-component, multi-user, and concurrent applications

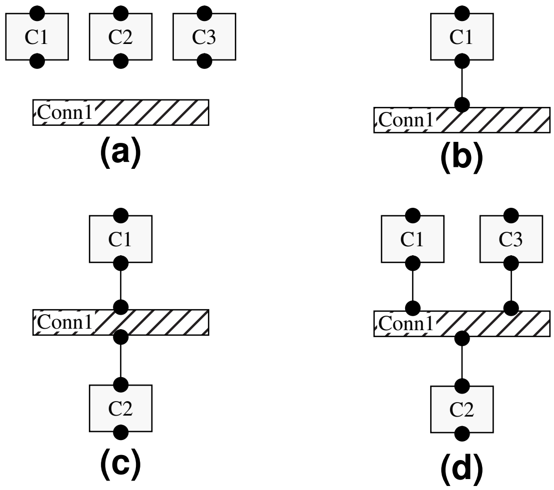

Fig. 2. C2 connectors have *context reflective interfaces*. Each C2 connector is capable of supporting any number of C2 components.
(a) Software architect selects a set of components and a connector from a design palette. The connector has no communication ports, since no components are attached to it.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

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 components to share threads of control. This separation of architecture from implementation is a key aspect of our approach to incorporating OTS connector 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 would 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 of 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.

```
C2Object
 ├──C2Message
 │   ├──C2Request
 │   └──C2Notification
 ├──C2Port
 │   └──C2Port_FIFO
 └──C2Brick
     ├──C2Connector
     │   ├──C2Connector_SameProcess
     │   ├──C2Connector_Thread
     │   └──C2Connector_IPC
     └──C2Component
         ├──C2Architecture
         └──C2Component_Threads
             └──C2Architecture_Threads
```

Fig. 3. C2 object-oriented class framework.

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 [12, 14].

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, shown in Fig. 3. This framework is the basis of development and OTS component and connector reuse in C2. It 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.

The framework has been implemented in C++ and Java; a subset is also available in Ada. The size of the framework in both C++ and Java is approximately 3000 commented source lines of code.

3.3 An Example C2 Application

The architecture that was used as the basis for our investigation of OTS connector 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

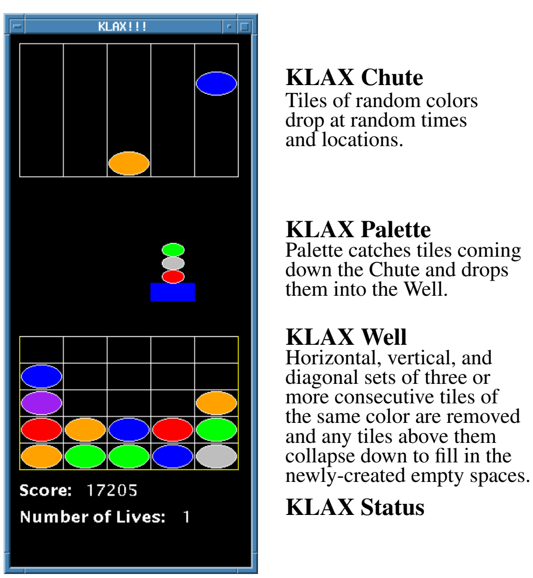

Fig. 4. A screenshot and description of our implementation of the KLAX video game.

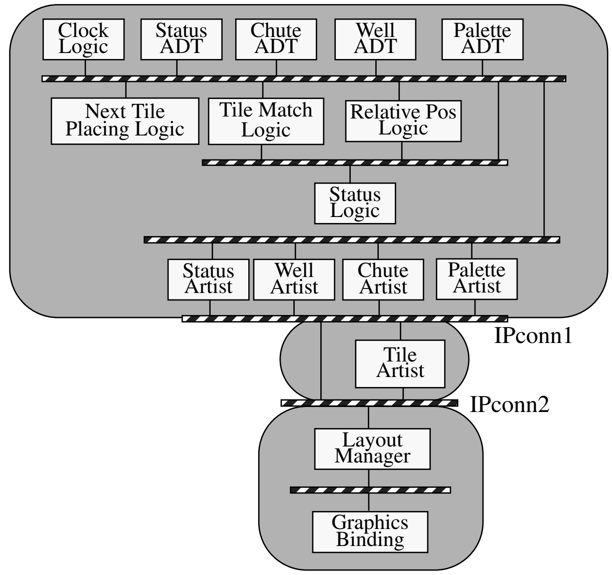

Fig. 5. Conceptual C2 architecture for KLAX. Shaded ovals represent process boundaries in the three-process implementations of KLAX.

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.

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.

Two KLAX implementations were built using the C++ and Java frameworks. Both implementations consist of approximately 8000 lines of commented code, in addition to the base framework's 3000 lines of code. Performance of both implementations 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 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 APPROACH

This section discusses the approach we took to integrating the OTS connector technologies and the dimensions along which each integration is evaluated. Although our initial intent was to devise both the integration strategies and the evaluation criteria *a priori*, they were also partly guided by our experience.

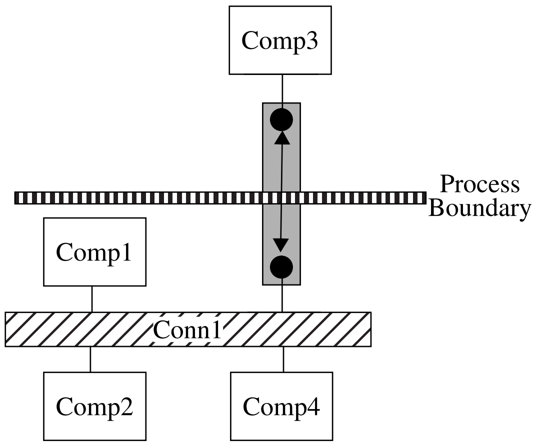

Fig. 6. Two communication ports in separate processes comprise a single "virtual port." For clarity, we do not highlight component and connector ports.

4.1 Possible Integration Strategies

We examined four possible approaches to connecting several parts of a C2 application running in multiple processes, and possibly on multiple machines. Two of the strategies divide an architecture along a connector, as indicated in Fig. 5, while the other two do so along a single communication port.[1] Three of the four approaches were successfully used.

4.1.1 Linking Ports across Process Boundaries

The first approach involved linking two C2 ports across a process or machine boundary. The single-process implementation of a C2 connector links two ports together by having each port object contain a reference to the other one. In this way, the ports can call methods on each other, sending messages as method parameters. Our intent was to simply exchange port references *across* process boundaries and use the same technique for message passing.

This strategy failed for several reasons. Most importantly, ports were not easily serializable. Any objects sent over the network 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. Thus, even if we could overcome the serialization issue and pass port objects through the network, a connection made between by using the references to them would not be preserved over the network.

Given the manner in which data is serialized and sent over process boundaries, we refined our original approach with C2 ports as the primary vehicle. Rather than attempting to send whole C2 port objects across process 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.

We built additional infrastructure to allow a process to join a C2 application at runtime. This was accomplished by creating separate network-enabled routines that could add ports to connectors based on a remote request during the execution of an existing architecture. New processes could join this existing architecture by connecting one of their own interprocess ports to the newly added port in the existing architecture.

4.1.2 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 architecture to the appropriate abstraction — the connector. In order to remedy this, we have also devised two connector-based approaches. Both of these approaches consist of implementing a single conceptual connector with two or more actual connectors, linked across process boundaries. Each actual connector thus becomes a segment of a single "virtual connector."

We call the first approach "lateral welding," depicted in Fig. 7a. Messages sent to any segment of the C2 virtual

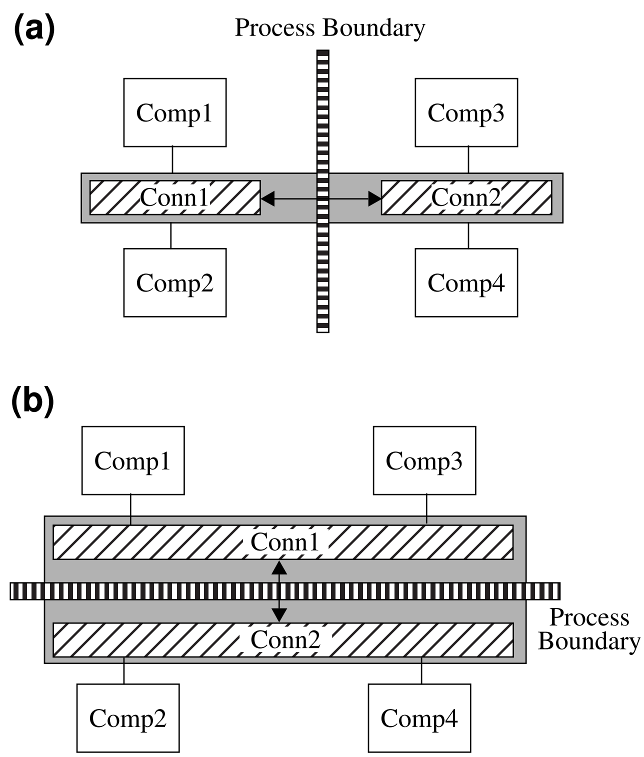

Fig. 7. Connectors as a primary vehicle for interprocess communication. A single conceptual connector can be "broken up" vertically (a) or horizontally (b) for this purpose.

1. Ports are intermediary objects that handle message pathways between connectors and components.

connector are broadcast to its 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. Depending on the OTS interconnection technology used to implement laterally welded virtual connectors, it may be possible to create and destroy these bindings at runtime, fundamentally changing the nature of a multi-process C2 application during its execution.

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. 7b. 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 interprocess communication medium.

4.2 OTS Technology Evaluation Criteria

When evaluating the OTS connector technologies and their C2 integrations, we have focused on several factors. It is unlikely that any one technology would satisfy all of these requirements. Therefore, the selection process must also be based on the characteristics and needs of a given application:

- *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 multi-lingual 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.
- *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*. If a given technology supports only interprocess communication, its utility is limited and additional types of connectors will need to be employed. Note that multiple types of connectors in an application may indeed be preferable, as each may optimize a particular type of communication.
- *similarity to 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. 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.
- *ease of integration and use* — if a given OTS technology requires a substantial amount of effort to integrate it into the implementation infrastructure and/or use it in an application, 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 integration with C2, since C2 mandates that all components execute independently of each other. The amount of work required to integrate Q, Polylith, and RMI (see Section 5) was relatively minor, typically 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 interconnection tools 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. An interconnection 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 3.3, several hundred messages may be generated every second. The ability to efficiently ferry those messages among the components and across process boundaries is paramount.

5 INTEGRATIONS OF OTS INTERCONNECTION TECHNOLOGIES

5.1 Q Integration

The Q system [8], developed at the University of Colorado, is intended to provide interoperability support for multilingual, heterogeneous component-based systems. Its intended use closely matches C2’s needs and Q was thus chosen as a candidate connector technology. Q supports systems built in several languages: C/C++, Ada, Java, Tcl, Lisp, and Prolog. Q was originally built for the UNIX platform, although its Java interface presents 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.

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 marshalling and unmarshalling 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.

Since one role of C2 connectors is as interprocess boundaries, 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 easily added in a Q-C2 connector.

A Q-C2 connector exports the same interface as a regular C2 connector, so that 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" horizontally into two or more Q-C2 connectors, one per process, as shown in Fig. 7b. 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, has to 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 as 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 application execution 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 also easily exceeded human reaction time if the *ClockLogic* component used short time intervals.

5.2 Polylith Integration

The Polylith software bus was developed at the University of Maryland [17]. 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 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 unmarshalling 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 from the bus.

Like the Q-C2 connector, the "Polylith-C2" connector is an extension of the standard, in-process C2 connector. The Polylith-C2 connector was implemented as an instance of the "lateral welding" approach shown in Fig. 7a. 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. 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. This limitation became problematic when Polylith-C2 was used in the implementation of the KLAX application from Section 3.3. 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 Integration

Java's Remote Method Invocation (RMI) [22] 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. These concepts are shared by other OTS interconnection technologies, including Microsoft's DCOM and OMG's CORBA. Because RMI is object-oriented, it fits comfortably within C2's object-oriented implementation framework. 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. 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 was integrated with C2's Java framework using both the virtual port approach, shown in Fig. 6, and the lateral connector welding approach, shown in Fig. 7a.[1] We attempted both variants of the virtual port approach, described in Section 4.1.1. As discussed in Section 4.1.1, passing port references via RMI was unsuccessful: unlike a standard Java method call, which passes object parameters by reference, all remote method calls pass their parameters by copy. The RMI-C2 connector has all the capabilities of single-process C2 connectors. Additionally, it has the ability to register itself with the Java-RMI name server, and to be linked to other registered connectors.

As Java is multithreaded, RMI is well suited for a multithreaded environment. It allows communication among objects running in different processes, which may be on different machines. Communication occurs exclusively over the TCP/IP networking protocol.

Minimal modification was required to convert an existing C2 application into a multi-process application that uses RMI-C2. RMI supports application modification at runtime, a capability enabled by Java's dynamic class loading. The performance of the three-process implementation of KLAX using RMI-C2 was satisfactory.

RMI's properties make it ideal for use within a Java C2 application. Its native support in Java 1.1 makes it somewhat easier to use than third party alternatives. Also, using RMI-C2 connectors does not preclude a C2 application implemented partially or completely in Java from using another connector technology, such as Q, as well.

6 CONCLUSIONS AND FUTURE WORK

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.

Another key aspect, and the focus of our work, is the ability to evaluate and integrate existing software interconnection technologies for use with software architectures. Integrating such technologies with architectural toolkits has the potential to expand the palette of supported prefabricated application building blocks (i.e., components). For example, the construction of a CORBA-based connector would enable one to utilize third-party CORBA components in an application.

To date, we have been able to successfully integrate three OTS connector technologies into our architecture implementation infrastructure. In the process, we have developed a set of integration techniques that show potential for general applicability.

At the same time, this is an ongoing project and many issues remain items of future work. These include the integration of other technologies, specifically ILU (Inter-Language Unification), a CORBA relative developed at Xerox PARC [24], and DCOM (Distributed Component Object Model), from Microsoft [20]. We also plan to continue refining our existing integrations to improve their performance and capabilities. Ultimately, our goal is to be able to simultaneously use several technologies within a single C2 application. For example, in an application, several UNIX processes may be bound with Q; they would use RMI to communicate with the portion of the application on a Windows machine, whose processes, in turn, communicate via DCOM.

7 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.

8 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. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, November 1997.
4. D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing, 1993.
5. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
6. A. Julienne and B. Holtz. *Tooltalk and Open Protocols: Inter-Application Communication*. SunSoft Press/Prentice Hall, April 1993.
7. D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, September 1995.
8. 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.
9. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of

1. We refer to the result of lateral connector welding as the "RMI-C2 connector."

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.

10. N. Medvidovic and R. N. Taylor. Exploiting Architectural Style to Develop a Family of Applications. *IEE Proceedings Software Engineering*, October-December 1997.
11. 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.
12. 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.
13. D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, Redmond, WA, 1996.
14. 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.
15. R. Orfali, D. Harkey, and J. Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Inc., 1996.
16. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.
17. J. Purtilo. The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, January 1994.
18. S. P. Reiss. Connecting Tools Using Message Passing in the Field Environ*ment. IEEE Software, pages 57–66, July 1990.*
19. A. Schill, editor. *DCE — The OSF Distributed Computing Environment*. Proceedings of the International DCE Workshop, Karlsruhe, Germany, Springer Verlag, October 1993.
20. R. Sessions. *COM and DCOM: Microsoft's Vision for Distributed Objects*. John Wiley & Sons, New York, NY, 1997.
21. 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.
22. Sun Microsystems, Inc. Remote Method Invocation. http://java.sun.com:80/products/jdk/rmi/index.html
23. 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.
24. Xerox Palo Alto Research Center. ILU — Inter-Language Unification. ftp://ftp.parc.xerox.com/pub/ilu/ilu.html