


• • •  
The growing need for systems that can integrate new and legacy components inspired development of this architecture for distributed components. The authors describe two implementations of the architecture and a sample application, one using Java RMI and the other using a Corba-based product.  
• • •



# An Integrated Network Component Architecture

**Israel Ben-Shaul**, Technion

**James W. Gish and William Robinson**, GTE Laboratories

•  Most corporate enterprises have a huge investment in software systems that support their diverse business processes. The National Institute of Standards and Technology estimates that 85 percent of installed software consists of custom applications in large organizations.<sup>1</sup> Yet most applications are monolithic and single-purpose, which makes them expensive to build and error-prone. In industries where systems must be deployed rapidly to retain a competitive edge, simplifying and automating the synthesis of software systems from reusable components proves essential. Hence the recent interest in component-based software engineering among researchers and practitioners.<sup>2-4</sup>

Our work explores architectures, technology, and products for building large-scale software systems, particularly highly interactive, distributed applications such as telecommunications customer support. In developing the Integrated Network Component Architecture, we sought to create an architecture with simple mechanisms for creating component interconnections. We believe that in the future, open,

flexible systems will be based on configurations of loosely coupled, distributed components—but these systems still must be scaleable, robust, and manageable. It should be possible to decompose an existing system into coherent components, some of which may be composed together with

architecture without considering how to implement it using existing technology can lead to mismatches between the architecture's conceptual model and the object models inherent in existing products. This mismatch, in turn, leads to unacceptably long development cycles because the architecture's novel components must be built before constructing the application content. We focus, therefore, on architectures that can be implemented atop today's foremost distributed object frameworks—Corba, DCOM, and Java RMI.

## **A software architecture should be realizable using available technology and products that support interoperability.**

other components in new applications. Meeting these goals requires rigorous application of sound architectural principles and an appropriate mating of the system architecture with available technology and products.

### **DEFINING SOFTWARE ARCHITECTURE**

As the literature's many alternatives<sup>5,6</sup> attest, the software engineering research community has not agreed on a definition of software architecture. Most definitions include the triad of components, connectors, and some attribute related to constraints, rationale, or properties. Architecture definition languages such as Rapide<sup>7</sup> formally define an architecture's components in terms of their interfaces and connections to one another. Many definitions exist for "component" as well, and most do not differentiate between components that are objects, programs, COTS products, or something else. An architecture is a conceptual model of how a system's components are defined and developed, and how they interoperate. We distinguish this from an infrastructure, which is the implementation of the conceptual model.

A software infrastructure, assembled from infrastructure systems and products, must conform to the architecture's constraints. For example, a simple client-server architecture includes user interface technology and a database management system. Specific infrastructure products include X Windows and Oracle DBMS. The infrastructure consists of these products and the components required to interconnect them.

A software architecture must be realizable using available technology and products that support interoperability. Choosing the latest technology and products does not guarantee a coherent architecture. On the other hand, defining a coherent archi-

### **INCA**

The Integrated Network Component Architecture builds connection mechanisms into the components and implements them according to the architecture. Although this constrains the set of components that can be composed, these constraints actually facilitate the usually difficult composition process for components that adhere to the architectural rules. We do not define architecture-independent mechanisms for universal component interconnection. See the boxed text "Defining Software Components" on page 83 for other approaches.

INCA's conceptual model defines four major elements: components, remote events, a runtime component configuration tool, and an interaction manager.

#### **Components**

Component granularity is a major concern in component-based architectures. INCA components are coarse-grained, encapsulating a business function such as listing the telecommunication services available to a given customer. Conventional fine-grained components intended to be reused arbitrarily are often too small for practical reuse since they require considerable work from application programmers. A component in INCA consists of one or more parts:

- ◆ a mandatory server that performs the business function (for example, Provision Long Distance),
- ◆ an optional user interface that lets users interact with the business function, and
- ◆ an optional back-end database access service that encapsulates access to system databases.

Figure 1 shows an abstract configuration of components. A full-blown single component follows the three-tier client-server subarchitecture. Components

are loosely coupled through event notification among component servers. This horizontal composition<sup>8</sup> approach follows the application's logical business functions, as opposed to vertical composition based on physical colocation of functionality. The infrastructure provides vertical connectivity<sup>9</sup> through "glue" system components that facilitate the composition of applications from existing components, with minimal impact to other components.

### Remote events

Servers belonging to different components communicate primarily through event notification, although non-event-based synchronous communication is still available when needed. A component server can receive notification of specific events that another component server might generate. The event-generating component server is the source; the receiving component server is the listener. Since a listener may be remote, we need a reliable mechanism to transfer the event from source to listener over the network.

One solution would be to create a centralized message server to which a source sends its events. The message server then dispatches events to listeners, similar to the Field approach.<sup>10</sup> This will not work for distributed components, however, since communication to and from the server and its execution load might create a bottleneck and a single point of failure. To avoid this bottleneck, INCA decentralizes the control flow for event notification.

To ease component interconnection, we defined a generic method for passing events and a generic event class from which all component-specific events are derived. One event proxy type exists for each component type. Each component that initiates an event loads an event proxy object for any other component to be notified when the event occurs. The proxy object is instantiated with the receiving component instance's location and logical name. When dispatching an event, the event source hands the event object to each event proxy object in its listener list. The event proxy propagates the event to the remote component it represents.

Figure 2 shows an event source, Server A, with a listener list containing proxies for three components:

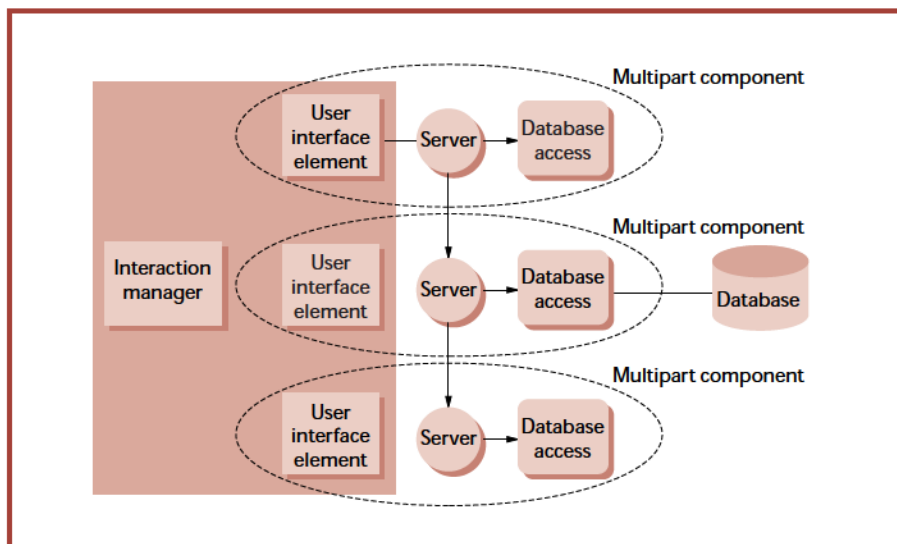


Figure 1. Multipart components in INCA.

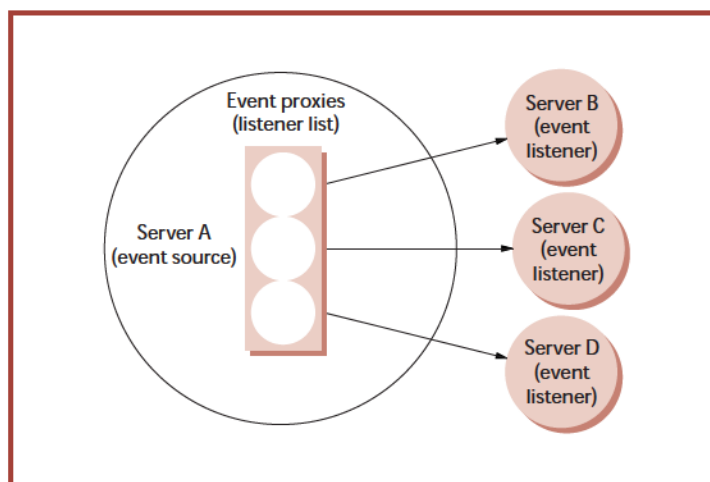


Figure 2. Remote events in INCA.

Servers B, C, and D.

### Runtime configuration

The architecture establishes component connections at runtime rather than at compile time or design time (as in the JavaBeans model). One component connects to another after one or both have been instantiated and deployed. Such dynamic configuration lets us customize applications while they are running and contributes to their

- ♦ availability, since there is no need to bring down a component for upgrades;
- ♦ reliability, since a crashed or faulty component can be replaced in the application; and



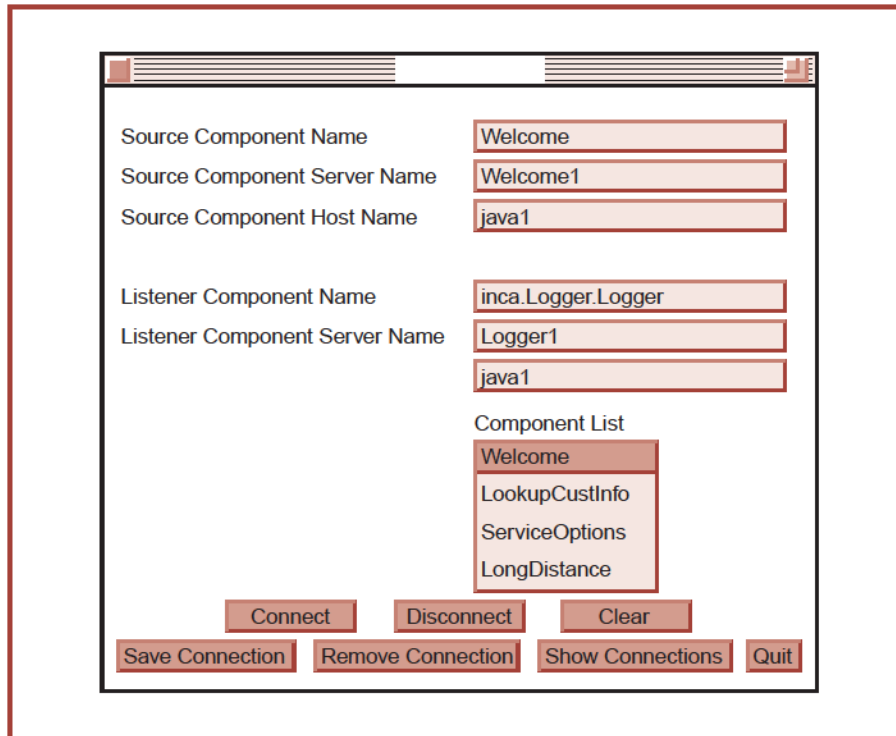


Figure 3. Configuration tool interface.

◆ performance, since component connections may be reconfigured as a result of changes in processor loads or network congestion between components.

The architecture defines a component configuration tool that lets an administrator establish event registrations among server objects. This tool links an event's source with the listener through proxies dynamically bound to the source servers. INCA also lets an administrator preconfigure the source-listener relationships in per-server configuration files that the servers read at startup. Since no centralized database of connections is stored in the runtime configuration tool, the tool is stateless and does not participate in event transmission. This approach has the advantages of late binding without the disadvantages of centralized control.

Figure 3 shows the configuration tool's user interface. To establish a new connection, the administrator specifies the component name, instance name, and location for both source and listener. The source then loads the proper proxy class and instantiates a proxy object (using Java's dynamic class loading and reflection, respectively). The proxy class name is formed by concatenating the component name with the string "Proxy" (for example,

LoggerProxy). The remote component need not be active when creating the proxy (requiring this would imply a tight coupling at runtime).

### Interaction manager

To compose components horizontally, with each possibly providing its own user-interface element, a vertical architectural element must combine the various clients into a single application. INCA defines an *interaction manager* that permits controlling the application's flow. It also serves as a clear target for server messages that affect the user interface, accepting control messages from component servers and invoking component clients. The interaction manager creates globally unique session identifiers, launches the first component, provides remote services for servers to launch subsequent

components, and handles component shutdown. Each workstation that runs an INCA-based application will have one instance of the interaction manager.

## APPLICATION PROTOTYPE

We chose the Service Provisioning Scenario to test the architecture. This well-known telecommunications scenario simulates a session between a customer and a phone rep. The Welcome screen takes the customer phone number and triggers a lookup of customer data. Once presented with a list of available services, the customer can request a new service, such as Long Distance. Upon completion, this interaction leads to creation of a new Service Request object. When the new service has been created, the provisioning server updates the customer data with a new service object and notifies the phone rep (and possibly the user) by e-mail.

We identified five components for the Customer Provisioning scenario:

◆ *Welcome*. This component provides a user interface through the WelcomeApplet, which accepts a customer's primary phone number and passes it

on to the WelcomeServer. The WelcomeServer then raises the CustomerArrivalEvent.

◆ *Customer.* When it receives the CustomerArrivalEvent, this component does a database lookup to retrieve detailed customer information. It then asks the AppletManager to launch the CustomerApplet that displays this data. At the customer's request, the CustomerApplet notifies the server that the customer wants to look at ServiceOptions. The server then generates the ServiceOptionsRequestEvent.

◆ *ServiceOptions.* Upon receiving the ServiceOptionsRequestEvent, the ServiceOptionsServer requests display of service options available for this customer, such as Long Distance or Call Waiting. When the user selects a service option, the user interface calls the server with the option, generating a ServiceOptionSelectedEvent.

◆ *Long Distance.* Upon receiving the ServiceOptionSelectedEvent with the choice "Long Distance," the server generates a list of carriers and asks the appropriate AppletManager to launch the LongDistanceApplet that displays the list. When the user selects a carrier, the LongDistanceApplet notifies its server, generating an LDSelectedEvent. The LongDistanceServer then adds a new service request object to the customer data through the Customer server.

◆ *LDHandler.* When it receives the LDSelectedEvent, this server contacts the appropriate carrier. After it receives confirmation of provisioning, it sends e-mail to the phone rep and the customer (if the customer has e-mail), and updates the customer's data to show that the ServiceRequest is closed and to add a new ActiveService object.

These are shown in Figure 4 (on the next page). We also defined a generic Logger component that listens for any events a server object emits and logs them to a database. Such a component can be dynamically attached to any component to trace its activities, without affecting any other component in the system.

## TWO INCA IMPLEMENTATIONS

We implemented INCA's conceptual model atop Java RMI and Corba. The model's framework independence greatly reduces the difficulty of implementing INCA on these distributed object frameworks.

## DEFINING SOFTWARE COMPONENTS

In recent work by Eric Evans and Daniel Rogers,<sup>1</sup> Java applets interact with remote servers via Corba. The authors note that using Java applets helps solve some client deployment problems. This applies to our Java/Web implementation of INCA. Evans and Rogers also used Corba servers to address platform and language independence, but they did not attempt to generalize this to other distributed object frameworks. Our abstraction of architectural elements makes it straightforward to migrate the INCA implementation to various frameworks.

Robert Allen and David Garlan<sup>2</sup> take a larger view of composing components, moving the level of discourse from the programming language level used to the composition of systems through sophisticated connection mechanisms. They introduce the concept of component connectors, first-class objects embodying protocol translations that complement components' interface specifications. We focused on the programming model for components, building the capability to interconnect dynamically into the components themselves.

George Heineman<sup>3</sup> defines a framework for adaptable software components that seeks to illustrate techniques for making components more reusable. The framework is language- and architecture-independent. Heineman's key contribution is the idea of adapting components to new uses rather than merely customizing them by setting specific parameters, as would be the case in JavaBeans. Ultimately, adapting a component requires writing wrappers in code to expand or limit a component's functionality. However, the framework does not prescribe a generic mechanism for interconnecting components.

## REFERENCES

1. E. Evans and D. Rogers, "Using Java Applets and Corba for Multi-User Distributed Applications," *IEEE Internet Computing*, Vol. 1, No. 3, May/June 1997, pp. 43-55.
2. R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Trans. Software Eng. and Methodology*, July 1997.
3. G. Heinemann, "A Model for Designing Adaptable Software Components," *Proc. COMPSAC 98*, IEEE Comp. Soc. Press, Los Alamitos, Calif., August 1998, in press.

## Preserving INCA across implementations

We find it essential to clearly separate elements that constitute the architecture and those that constitute the infrastructure, especially when the architecture must be migrated over time from one technology to another.

We first decided to make the user interface elements applets so that they could be launched from a Web browser. This decision prompted us to use the Java programming language for our components' user interfaces and server elements.

At the architecture level, we defined a set of classes and interfaces that express INCA's essential concepts. At the infrastructure level, we mapped

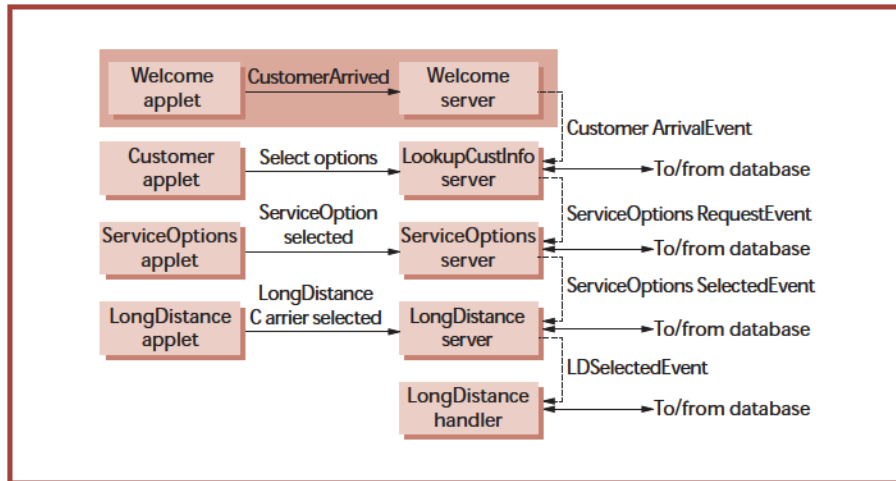


Figure 4. The prototype application: Components for customer provisioning.

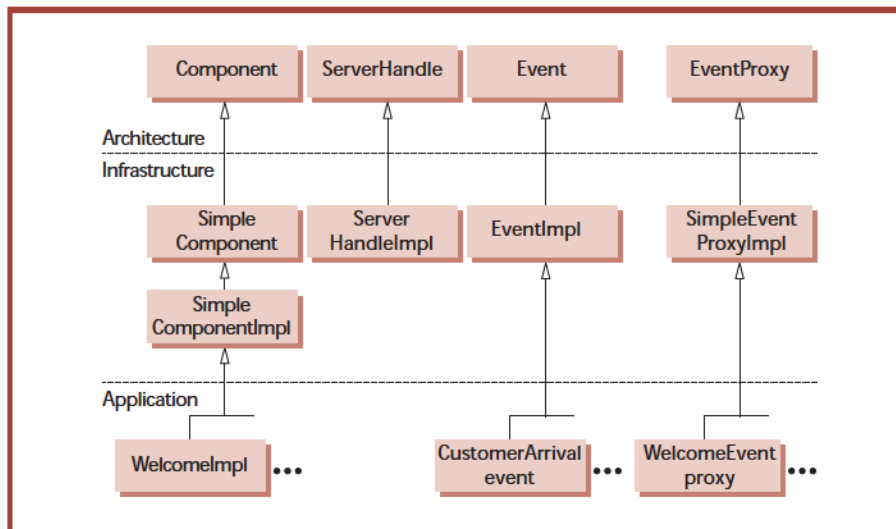


Figure 5. Generic server classes.

these architectural elements into specific implementations using facilities of each underlying distributed object framework. The distributed object framework should provide location transparency for reliable remote message delivery and naming services (registration of a remote object reference by name and lookup by name). At the application level, we populated the system with application-specific instances of components. Figure 5 shows the generic object model for the server elements at these three levels of abstraction.

The INCA server base classes include Component, ServerHandle, Event, and EventProxy.

◆ *Component* defines services for adding and removing listeners, and a callback to handle events sent to the component.

◆ *ServerHandle* defines three elements needed to locate and bind to a particular server: the component's name, the server's name, and the host name on which the server resides.

◆ *Event* defines the contents of a generic event. INCA events are typed, and each event object contains two levels of information. The first, which is application-independent, includes a SessionID identifying the session in which the event has occurred and a reference to the InteractionManager for the workstation that set the SessionID. The second level adds information specific to the application component.

◆ *EventProxy* is for event forwarding, and uses the underlying distributed object framework to invoke a listener's notification method. Its forward() method is an example of the Template Method design pattern.<sup>11</sup> The forward() method invokes a filter() method to eliminate any events not of interest to its listener, then binds its ServerHandle to a specific remote reference, and finally sends the event to the listener. Specific subclasses of EventProxy will

override the filter() method as needed.

As shown in Figure 5 we define simple implementations of those architectural elements specific to a given distributed object framework at the infrastructure level. In some cases, these implementations (for example, methods in SimpleComponentImpl that support interest registration) will suffice for most components, although a component server can override these methods if necessary. At the application level, each component server, such as Welcomelmpl, extends the generic implementation.

Figure 6 shows the comparable model for user interface elements, which is somewhat simpler



than the server side. The `InteractionManager` exists at the architecture level. At the infrastructure level, we define a concrete `AppletManager`, which is a specialization of the `InteractionManager` to the Web environment. The `AppletManager` is itself a remote, application-independent server. We also define a generic `SimpleApplet` class that knows how to find its `AppletManager` and how to gather up context information to pass on to its associated server. At the application level, subclasses of `SimpleApplet` define specific user interfaces and application logic.

Keeping these levels of abstraction separate lets us describe the implementation on each framework simply by describing where our generic classes fit into the framework's object model.

### The Java RMI implementation

Our first INCA implementation used Java RMI for remote message delivery and naming services. Each component server is implemented as an RMI remote object. A component client, if any, is implemented as an applet. Since browsers work in a stateless fashion, any state that must be preserved is sent to the applet's server subcomponent. A component's applet also holds a reference to its `AppletManager` object.

We implemented the runtime configuration tool as a simple Java application that manages the connectivity between active components by interacting with their internal built-in connector objects. We implemented the applet manager as a Java remote object.

To be a remote object, our component class must implement the `java.rmi.Remote` interface, the root of the Java remote object hierarchy. In addition, class `SimpleComponentImpl` must implement the `SimpleComponent` interface and extend the class `java.rmi.server.UnicastRemoteObject`. This latter class provides support for packaging an object as a remote object in a standalone application. Finally, for each application component, such as `Welcome`, we must define an interface that extends the interface component and defines the specific remote methods provided by the `Welcome` server. This interface is implemented in the class `WelcomeImpl`.

By following only a few conventions for remote objects, we created our server objects as remote objects. We generated the requisite stubs and skeletons for remote objects by running the compiled implementation classes through the Java remote method invocation compiler, `rmic`.

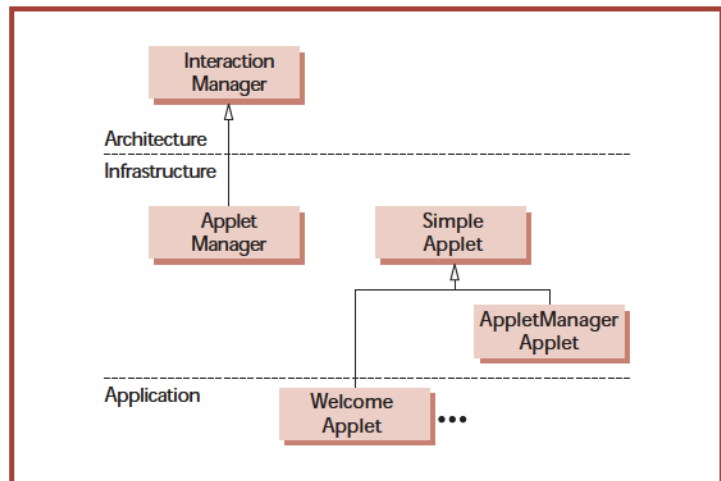


Figure 6. Generic user interface classes.

### The Corba implementation

We also implemented INCA using the Visigenic Visibroker for Java, version 3.2. Each component server and the `AppletManager` are implemented as Corba objects. Component applets are largely unchanged from the Java RMI implementation, the primary modification being how they deal with slightly different signatures and exceptions thrown by the remote methods in the servers.

All Corba objects, including our component class, are derived from the root class `org.omg.Corba.Object`. We wrote a set of IDL files that describe the interfaces for the architecture elements and the application components. These were fed into the Visibroker compiler `idl2java` to generate the appropriate stubs and skeletons, as well as some additional helper classes. The process is more complex than in Java RMI because the collection of generated classes is quite large.

## EVALUATION

Evaluating the INCA component model's architectural characteristics and contrasting the two implementations yields the following insights.

- ♦ *Component model.* The component granularity we adopted is somewhat novel. Each component is, in effect, a mini client-server package. This has worked well so far, but further (re)use of these components, especially in other contexts, will bear out our approach's long-term effectiveness. From a development viewpoint, once we had implemented the infrastructure's core pieces and the first

## COMPONENT TECHNOLOGIES AND PRODUCTS

For detailed specifications of Corba, Java RMI, DCOM, and JavaBeans, consult the following sources.

Object Management Group, "The Common Object Request Broker: Architecture and Specification," revision 2.2, Feb. 1998, <http://www.omg.org/Corba/CorbaIiop.htm>.

N. Brown and C. Kindel, "Distributed Component Object Model Protocol—DCOM 1.0 Internet Draft," Jan. 1998, <http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt>.

Sun Microsystems, Inc., Java RMI Specification, Internet Draft, Dec. 1997, <http://www.javasoft.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html>.

Sun Microsystems, Inc., JavaBeans Specification 1.01, July 1997, <http://www.javasoft.com/beans/docs/spec.html>.

Sun Microsystems, Inc., Enterprise JavaBeans Specification 1.0, Mar. 1998, <http://java.sun.com/products/ejb/docs.html>.

two components, it was easy to add new components without perturbing existing ones.

♦ *Distributed control.* INCA has no single component that controls the interactions among components. Each component listens to and acts on the events for which it is configured, and generates its own events and distributes them to its registered listeners. This distributes control throughout the system, hence there is no single point of failure. To avoid the loss of a component, which could cause application failure, the system can be configured with multiple instances of it.

♦ *Platform independence.* Component servers may run on any host and platform on which the Java Virtual Machine and the Corba software are implemented.

♦ *Location transparency.* Components may deliver events to other components regardless of the target components' location.

♦ *Browser-based user interface.* A Web browser provides an integrating platform for the component's client parts that require a GUI. Together, the Web browser and the INCA applet manager force component designers to adhere to a Web deployment model.

♦ *Administration.* The component configuration tool made it easy to reconfigure the connections among the event sources and their listeners for the components we built. However, server identity, which is used to locate a server listener, could become a serious administrative issue. With dozens or hundreds of server types running on many servers, the startup/shutdown procedures will need careful attention.

♦ *Event model.* Using events to loosely couple component servers generally proved successful.

We did find that, in the application scenario we described, events do not suffice for all communications among servers. Both the LongDistance and LDHandler components had to make a direct call to the customer server to update the database. Components that primarily function as database servers will likely have update and retrieval methods that one or more servers call directly. The key design problem this creates is making the database server instance known to the server that needs to invoke its methods. We solved this in our scenario by passing a reference to the customer server as part of the event data. But as the number of scenarios increases, event modeling at a larger scope becomes necessary—a fundamental concern for the system's scalability.

### Implementation issues

We experimented with several Web browsers, and found serious inconsistencies in how they handle Web page and applet management. This means an application cannot count on a browser to maintain the set of active applets required for a session. In particular, the AppletManagerApplet, which implements the interaction manager interface, may get purged prematurely, resulting in a downstream server's inability to launch its associated applet. The various Web browsers also employ different security policies, implement different Java release levels, and inconsistently support Java applets. We used only HotJava in this implementation.

We found the Java RMI implementation to be simpler than the Corba one, for two reasons. First, our implementation language is Java and the Java remote object model is a natural extension of the local object model; the language-independent Corba object model, however, must be mapped to Java, which leads to some odd mismatches. For example, in the Java RMI implementation, if a client holds a reference to a Java remote object that is of a more specific type, a simple downcast suffices to treat the object reference as a reference to the more specific type. On the other hand, in the Corba implementation, if the client holds a reference to a Corba object reference that is of a more specific type, the downcast will not work. Rather, in the client code, we had to use a "narrow" operation on the appropriate Helper class to effect the cast to a more specific Corba reference type. Choosing the "appropriate" Helper class was not always straightforward.

The second reason for Java RMI's simpler implementation had more to do with the nature of INCA



itself. In INCA, events are typed, and subclasses of the root event class add different kinds of structured data. In Java RMI's case, the event classes are all local classes, and the RMI mechanism will automatically pass these by value, as desired. We found that while Visibroker provides a preliminary implementation of "objects-by-value," the implementation is incomplete. For instance, if a Corba object reference was a member of the event structure, the ORB would not marshal it, nor would it marshal a Java class reference whose value was null. We found we had to bend our model a bit to work around the problem.

**T**his work advances our goal to establish a basis for experimentation in component-based architectures that support multiple interconnection and interoperability technologies. We are particularly interested in further INCA implementations using DCOM and Enterprise JavaBeans, and plan to test our components' viability in workflow-based applications. We also plan to explore operational requirements, such as configurability, administration, and monitoring. ♦

## REFERENCES

1. National Institute of Standards and Technology, "Focused Program: Component-Based Software," <http://www.atp.nist.gov/atp/focus/cbs.htm>, 1997.
2. DARPA, "Evolutionary Design of Complex Systems, Architecture and Generation Cluster," <http://www.sei.cmu.edu/~edcs/CLUSTERS/ARCH/index.html>.
3. O. Nierstrasz and T. Meijler, "Research Directions in Software Composition," *ACM Computing Surveys*, ACM Press, New York, June 1995, pp. 262-264.
4. I. Ben-Shaul et al., "HADAS: A Network-Centric Framework for Interoperability Programming," *Int'l J. Cooperative Information Systems*, Vol. 6, Nos. 3 & 4, 1997, pp. 293-314.
5. D. Perry and A. Wolf, "Foundations for the Study of Software Architecture," *ACM Sigsoft - Software Engineering Notes*, Vol. 17, ACM Press, New York, 1992, pp. 40-52.
6. D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering*, Vol. 1, World Scientific Publishing, Singapore, pp. 1-39.
7. D.C. Luckham and J. Vera, "An Event-Based Architecture Definition Language," *IEEE Trans. Software Eng.*, Vol. 21, 1995, pp. 717-734.
8. D. Barret et al., "A Framework for Event-Based Software Integration," *ACM Trans. Software Eng. and Methodology*, Vol. 5, No. 4, 1996, pp. 378-421.
9. I. Ben-Shaul and G. Kaiser, "Federating Process-Centered Environments: The Oz Experience," *Automated Software Engineering*, Vol. 5, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998, pp. 97-132.
10. S. Reiss, *The Field Programming Environment: A Friendly Integrated Environment for Learning and Development*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1995.
11. E. Gamma et al., *Design Patterns: Elements of Reusable Software*, Addison Wesley Longman, Reading, Mass., 1995.

## About the Authors



**Israel Ben-Shaul** is a faculty member in the Electrical Engineering Department at the Technion, Israel, and a consultant in the Information Retrieval Department at IBM's Haifa Research Laboratory. As a research staff member at Columbia (1990-1991) he also served as the manager of the Marvel project. Prior to joining Technion he was a research fellow at the IBM research laboratory in Haifa. His research interests include software architecture and interoperability, distributed and heterogeneous systems, Java and Internet computing, software engineering environments, workflow management systems, and advanced transactions and collaborative work.

Ben-Shaul received a BSc in mathematics and computer science from Tel Aviv University and an MSc and a PhD in computer science from Columbia University.



**Jim Gish** is a principal member of the technical staff at GTE Laboratories. He is engaged in the creation of a component-based service provisioning system using Enterprise JavaBeans and Corba. Since joining GTE Labs in 1987, he has worked on several software engineering research topics, including reuse, software understanding, process modeling, and software architecture.

Previously he was a principal developer and development manager of e-mail systems at Prime Computer & Datapoint. He is a long-time user and proponent of object-oriented programming languages, starting with his use of Simula 67 in 1974.

Gish received a BS in statistics and computer science from the University of Delaware and an MS in computer science from the University of Wisconsin-Madison. He is a member of IEEE.



**Bill Robinson** is a senior principal member of the technical staff at GTE Laboratories. He has over 20 years' experience teaching and practicing software engineering in a variety of R&D environments. Over the past 10 years he has been active in defining and developing distributed object management systems at Data General, HyperDesk, and Fidelity Investments. His research interests include architecture, design patterns, and distributed, component-based systems.

Robinson received a PhD in mathematics from Tulane University.

Address questions about this article to Robinson at GTE Laboratories, 40 Sylvan Rd., Waltham, MA 02035; e-mail [william.robinson@gte.com](mailto:william.robinson@gte.com).