Mark Astley, Daniel C. Sturman, and Gul A. Agha

CUSTOMIZABLE MIDDLEWARE FOR MODULAR DISTRIBUTED SOFTWARE

Simplifying the development and maintenance of complex distributed software.

A distributed system consists of a collection of autonomous computing elements that interact through a shared network. The asynchronous nature of distributed systems significantly complicates application development. In particular, software executing on distributed systems represents a unique synthesis of application code and code for managing requirements such as heterogeneity, scalability, security, and availability.

As an example, consider a distributed video-on-demand service: such an application requires policies for opening new connections to the service, ensuring that service providers are compensated (that is, charging for service), and managing the delivery of media to clients. While such policies express basic requirements for the service, the protocols that implement these policies are determined by orthogonal factors such as the presence of faulty hardware or insecure networks. These factors are orthogonal in the sense that they only affect the implementation of the protocol and do not alter the basic implementation of the service. As a result, changing requirements and the rapid pace at which new hardware is introduced lead to protocols that are constantly evolving. Application code and protocol code are often intermixed and this lack of modularity significantly reduces the flexibility, maintainability, and portability of distributed code.

We describe a system architecture and programming techniques that enable the clean separation of protocols from application code. In particular, we provide modularity via two mechanisms (see Figure 1):

- **Separation of Policies and Protocols:** Policies, which define the rules by which components interact (that is, client/server, transactions, publish/subscribe), are specified separately from protocols, which define the mechanisms by which policies are implemented.
- **Protocol Composition:** In cases in which a single component must adhere to multiple policies, a framework for protocol composition allows multiple policy implementations to safely coexist.

Separating policies and protocols provides an abstraction boundary in the spirit of abstract data types. Similar to the manner in which object-oriented programming realizes abstract data types by enabling the publication of a component's interface while hiding its implementation, the protocol abstraction presents an interface representing an interaction policy while hiding the details of the

Figure 1. Modularity is exploited at two levels. Interaction policies are specified from the protocols used for their implementation. Protocol composition allows components to participate in multiple policies.

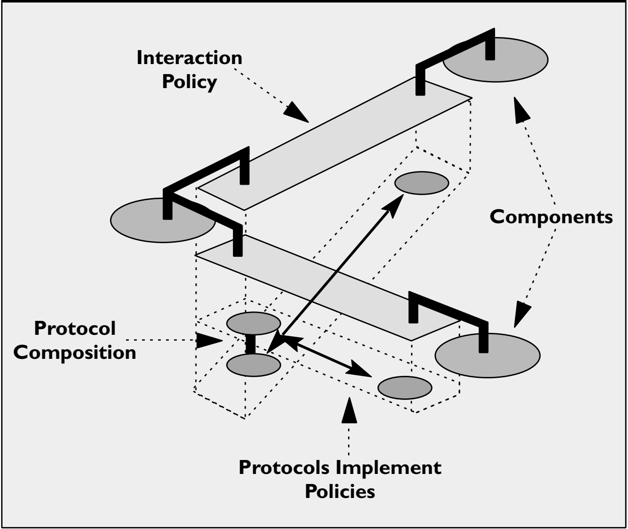
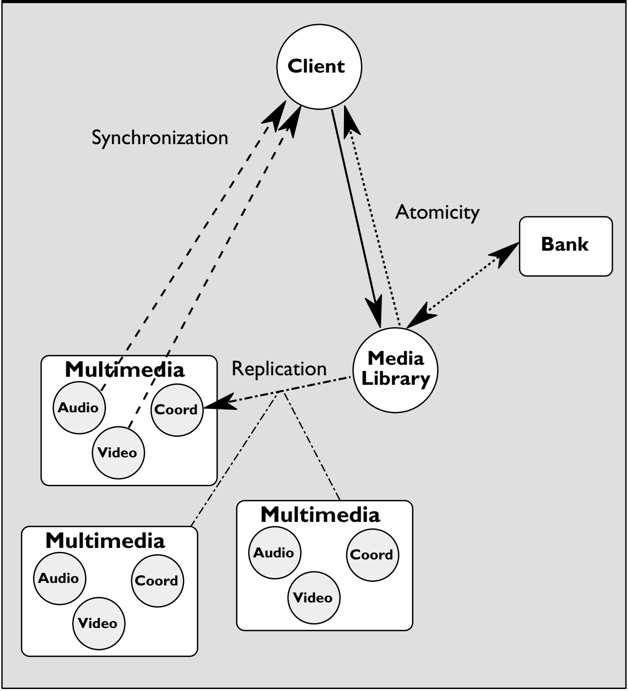


Figure 2. Video-on-demand is supported by multiple interaction policies including synchronization, atomicity, and replication (availability).



coordination mechanisms used to implement the policy. In particular, the protocols required for interactions need not be hard-coded in application code. This allows objects and protocols to be independently developed and later composed into runnable systems. For example, atomicity among a collection of objects may initially be implemented using a two-phase commit. If architectural constraints later change, for example requiring stricter reliability properties, modular design allows us to replace a two-phase commit with a three-phase commit without modifying client code.

Orthogonal design constraints may also force components to adhere to multiple policies. A framework for protocol composition preserves the modularity of such policies so that their implementations may be safely composed. For example, an existing encryption protocol may be safely composed with an existing primary backup protocol in order to satisfy both secrecy and fault-tolerance policies.

To illustrate our approach, we describe the Distributed Interaction Language (DIL). A DIL protocol is a linguistic abstraction that encapsulates protocol behavior: each protocol describes a customized response to events within the system. Distributed interaction policies are implemented by customizing system events defining component interactions (such as the sending and receiving of messages). DIL protocols dynamically modify system behavior: for example, in response to communication events, a protocol may add or remove messages, record or replace state, or halt the execution of a component. Events may also be programmer-defined. A combination of system and programmer-defined events together defines an interaction policy; the protocol provides an implementation for these events and, therefore, an implementation for the policy. The protocol (implementation) may be modified without affecting the events comprising the policy (interface).

A Distributed Application

Consider a distributed video-on-demand system (Figure 2). A client requests to view a movie from an online media library. The media library, in turn, communicates with the bank to validate the financial transaction paying for the client's request and delegates the multimedia server to transmit the movie. The multimedia server is itself comprised of multiple components and is replicated for fault-tolerance. However, replication is transparent to both the client and media library components.

This distributed application consists of four application components and many different interaction policies. We identify three policies that are representative of a broad class of distributed interactions. We observe that each policy may be implemented by multiple protocols, depending on the underlying hardware and operating system, and the requirements of the participating application components:

- **Atomicity:** Concurrency control to enforce distributed atomicity is necessary to ensure that financial transactions between clients, the media library, and the bank are atomic. Possible protocols for implementing this policy are two-phase commit or three-phase commit. A three-phase commit protocol survives crashes by the media library but requires additional message overhead.

Object-based Middleware

The term "middleware" refers to software technology that enables the modular connection of distributed software. Specifically, coordination in distributed applications is defined in terms of policies, which represent the rules component interactions must adhere to. The role of middleware is to abstract over the low-level protocols required to implement these policies. Typically, middleware applications are specified as if all interactions were local. The system itself is responsible for handling low-level details such as sending messages over the network, providing for synchronization, and guaranteeing reliability. This allows software developers to design applications that are "network transparent." That is, the application need not be explicitly aware of the composition and distribution of hardware.

Object-based middleware extends object-oriented programming to networked environments. Objects interact by invoking methods on other (possibly non-local) objects. In many implementations, middleware provides the illusion of local interactions by using remote procedure call (RPC) to intercept local method calls and translate them into invocations on the remote target (see the figure appearing in this sidebar). Typically, each object is given a stub, which is an interface describing the services exported by a remote object. A skeleton receives messages from remote stubs and translates them into invocations on a local object. Together, stubs and skeletons implement the appropriate set of protocols for carrying out a remote method invocation. Stubs are responsible for converting method arguments into a network transmittable form (called marshaling), sending the invocation request, and delivering any result to the calling object. Similarly, skeletons are responsible for decoding incoming requests, invoking the appropriate method on a local object, and transmitting a reply to the calling stub. Both CORBA and Java's Remote Method Invocation (RMI) implement the distributed object model (see www.omg.org/corba and ftp.javasoft.com/docs/jdk1.1.)

The current version of RMI only supports RPC semantics for method invocations. The most recent version of CORBA includes traditional RPC support as well as higher-level "services" that allow software developers to specify other policies for interactions. For example, publish/subscribe applications may be specified using CORBA's Event Notification Service. JavaBeans is a more ambitious approach, also based on RPC, in which applications are built by connecting "plug-in" objects through event-based interfaces. In terms of middleware, Beans are meant to be rather general distributed objects that may be placed and reconfigured in a wide variety of settings. While policies used at the application level remain fairly standard, the selection of protocols used to implement these policies depends heavily on the execution environment. For example, middleware designed for lossy networks will require reliability protocols in addition to any other mechanisms used to implement policies. Approaches such as CORBA and RMI provide little or no support for modularity at the level of protocol selection. As a result, software developers are often forced into ad hoc solutions in order to satisfy application requirements. A key challenge for future middleware developers is to provide for modular specification and customization of protocols as well as policies. Research is now beginning to address such concerns (for example, see [12]). C

An example of middleware support for RPC. The protocols that implement RPC as well as those that satisfy other requirements (for example, security) are hidden from clients and servers.

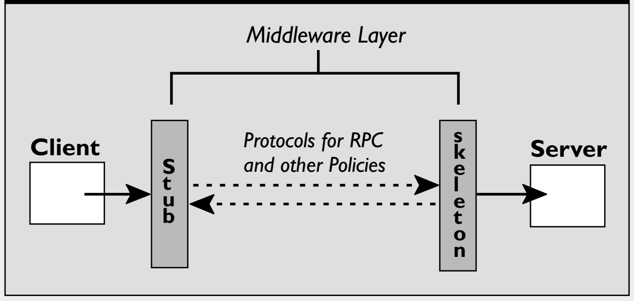
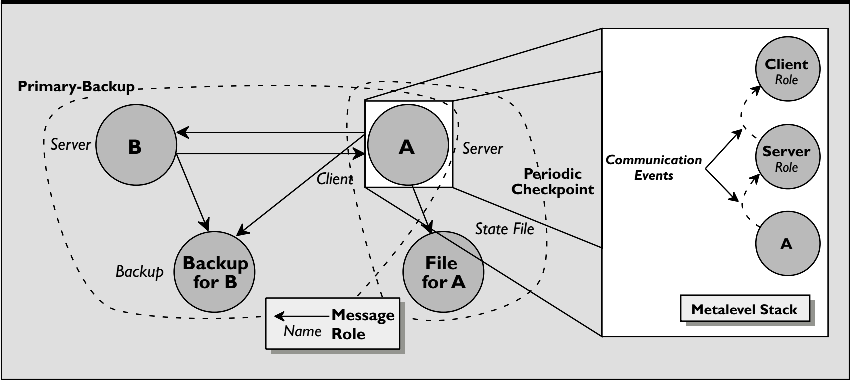


Figure 3. Two protocol instances whose set of participants overlap. In the primary-backup protocol, component A is in the role of client. In the periodic checkpoint protocol, component A is in the role of server. Protocol instances are organized into a metalevel stack.



the client from the multimedia server must be synchronized (lip-synched) so that audio and video frames are paired correctly. This requirement suggests the use of a real-time synchronization protocol. By sacrificing some accuracy, the two data streams are kept synchronized while satisfying the real-time requirements of the viewer. In a system without such stringent real-time requirements, a protocol with stronger correctness guarantees may be substituted.

- **Availability:** The multimedia server is replicated to provide fault tolerance. Many replication protocols exist that vary the failures they protect against, recovery time after a failure, and their cost in resources. For example, if the multimedia servers exhibited fail-silent semantics, a primary-backup protocol would provide fault-tolerance without excessive cost in computational resources. If recovery time is critical, an active replication protocol consuming additional resources would be used.
- **Synchronization:** Audio and video received by

Each component is governed by a set of interaction policies. For example, interactions with the multimedia server involve a replication protocol. Also note that the same component may have different policies governing interaction with different components. For example, the media library has different policies governing its interactions with the multimedia server, the client, and the bank. Thus mechanisms supporting protocol implementation must allow customization on a per-component or per-interaction basis. By examining the protocols implementing the interaction policies in this exam-

Actors

Conventional objects encapsulate a state and a set of procedures to manipulate the state. Objects provide an interface defined in terms of the names of procedures that are visible. These procedures, called methods, manipulate the local state of the object when invoked. In particular, this implies that representations supporting the same functionality may be interchanged transparently.

We model components of a distributed system as collections of concurrent objects based on the Actor model [2]. Actors extend the object model by encapsulating a thread of control together with the state and set of procedures. Actors are a powerful modeling device: memory chips, control devices, actuators, programs, and entire computers can be uniformly represented as actors. Moreover, the Actor model provides a mathematical foundation for reasoning about dynamic creation and reconfiguration in open distributed systems. The use of actors does not require a commitment to a particular programming language or syntax: because a distributed system can be naturally decomposed into several autonomous entities that communicate with each other, the Actor model provides a formalism for representing arbitrary distributed systems. Communication between actors is asynchronous. Asynchronous communication preserves the available potential for parallel activity: an actor sending a message asynchronously need not block until the recipient is ready to receive (or process) the message. Furthermore, more complex communication patterns such as RPC or real-time data delivery, the latter being particularly important for the multimedia example described in this article, are readily expressed within the Actor model [2, 9]. Actors may also create new actors. Actor creation returns a new mail address, which serves as an end point for actor communication. The programming model used in mobile agents and intelligent agents, two currently active areas of research, also satisfies Actor semantics (see [8]). C

Figure 4. Code for the two-phase commit protocol: the component coordinating transactions will have the role of *initiator*; components voting on transactions will have the role of *server*.

```
protocol Two-Phase {
    initialize(actor init)
        init actor assumes initiator role.
    addServer(actor svr)
        Add another actor in the server role.
    role initiator {
        MsgList currentAction;
        event beginAction()
            Begin construction of message list.
        event endAction()
            Close message list.
            Send "prepare" to servers.
        event transmit(Msg m)
            Add message to list (if being built).
      votePrepared(actor resp)
            Record prepared response.
            If complete, commit transaction.
        voteAbort()
            Abort transaction at all servers.
        committed(actor resp)
            Record commission.
            If complete, discard transaction.
 }

role server {
    MsgList currentAction;
    prepare(MsgList ml)
        Record message list and invoke query
            event to check acceptance.
        Reply with prepared or abort message.
    commit()
        Commit transaction and deliver message list.
    doAbort()
        Abort transaction by invoking abort event.
  }
}
```

ple, we identify the facilities necessary to allow separate development of protocols and applications.

Mechanism to Enforce Protocols

In order to better reason about distributed communication, we adopt a uniform programming model for distributed components. Components are limited to interaction solely through (asynchronous) message passing; we view components as encapsulated entities that do not share state. Communication is point-to-point and components may only communicate with known acquaintances. This view of a distributed system is formalized by the Actor model of computation.

We exploit the concept of reflection to isolate the communication behavior of a component. Reflection provides an explicit representation of system behavior in the form of metaobjects. Applications may be customized by replacing system metaobjects with user-defined metaobjects. In particular, a protocol may be expressed as a collection of metaobjects customizing communication over a group of application components. Each metaobject customizes communication for a single component. A protocol specified over a group of components is implemented collectively by the metaobjects customizing each component.

Moreover, by viewing metaobjects themselves as application components we allow further customization by way of meta-metaobjects. That is, metaobjects that customize other metaobjects. We use this mechanism to implement protocol composition. Specifically, multiple protocols may be applied to a single component by "stacking" the metaobjects that implement each protocol. Thus, a metaobject customizes the metaobject or component immediately below itself in the stack.

Interactions between application components and metaobjects are modeled in terms of events. Events define the interface between an application and a protocol. Both applications and metaobjects generate events to request specific services (for example, communication). Metaobjects define a default behavior for communication events:

- `transmit`: A component sends a message asynchronously. The event is parameterized by one argument: a structure representing the message being sent. The message structure may be copied, stored, destroyed, modified or examined in any way. By default, this event is processed by sending the message to its destination.
- `deliver`: A component receives a message. The event is parameterized by one argument: a structure representing the message being delivered. A metaobject may maintain a mail queue to store delivered messages until they may be dispatched. By default, this event is processed by queuing incoming messages and dispatching them in the order of their arrival.
- `dispatch`: A component is ready to process a message. Normally, a metaobject selects the next message to be processed and delivers it to the base object. The base object may be delayed if no message is available for delivery. By default, this event is processed by returning the next message on the queue constructed by `deliver`.

Metaobjects may provide an alternative behavior for these events in order to customize communication. Additional events may be defined by the programmer for explicit invocation in an application. For example, `beginAction` and `endAction` are metaobject events that allow an application to start and end a transaction.

Similarly, application components define a default behavior for state manipulation events:

- `reifyState`: Generated by a metaobject to request an encapsulated representation of component state. The internals of this representation may not be accessed or modified.
- `reflectState`: Generated by a metaobject to replace the state of a component with a new state (for example, with one returned from a previous call to `reifyState`).

These events are used by metaobjects in order to query or modify the state of a component. As in the case previously described, additional events may be defined by the programmer to support specific policies. For example, query and abort are application events that allow a metaobject to determine the commit status of an application.

Protocol Description

Using reflection as the mechanism, we have developed DIL: a language for high-level specification of protocols. A protocol is a single linguistic abstraction detailing an implementation of a distributed interaction policy. In DIL, a protocol is defined in three parts: parameters, role definitions, and protocol operations.

Protocol parameters are a set of global values that are shared by all participants in the protocol. Since sharing of distributed memory is nontrivial, parameters may only be assigned when the protocol is initialized.[1] Parameters enable further customization of protocols. For example, a replication protocol may use a parameter to specify the number of replicas in the protocol. A protocol using failure detection may have a parameter representing the time-out value between liveness checks.

Roles define the customized behaviors assumed by participants in the protocol. To implement a protocol, a metaobject is created to customize each participant. The behavior of the metaobject is determined by the role assumed by a component. When a role is assumed, the assuming participant's communication behavior is modified using a metaobject. The modified behavior applies only to interactions with other participants in the protocol (that is, those components that have assumed a role in the protocol).

Protocol operations govern installation of roles and other global actions on the protocol itself (not the individual participants). For example, initialization of a protocol is an operation invoked when the

[1]Consistency in distributed shared memory is an example of a distributed interaction policy. Hard-wiring a particular implementation of the policy into DIL would be counterproductive to the generality we are trying to provide.

Reflection

Traditionally software has been organized into two levels: application and system. Applications interact with the system through system calls, thereby invoking a predefined function at the system level. To extend this model to support customization, micro-kernels move much of the functionality reserved for the system into the application domain [1]. Object-oriented operating systems also support customization through the use of frameworks: sets of classes customizing the operating system for a particular execution environment [5]. These approaches, however, result in coarse-grained customization of a system. In many cases, optimization of an application requires customization on a per-object basis. Reflection provides this ability: it allows application objects to customize the system behavior that describes their own behavior [10].

Reflection is realized through manipulation of system (or meta-) level objects that implement some functionality in an object's behavior. For example, metalevel objects may customize memory allocation on a per-object basis. Traditionally, an object allocates memory by making a system call. In a reflective system, an object may allocate memory by invoking a customized metalevel memory object. Possible customizations include improving continuity of memory values or supporting automatic garbage collection. Invocation of the metaobject is completely transparent to the application: code for the reflective and nonreflective systems is identical. Architectures for reflective customization may be quite general. For example, an interpreter-based approach may be used where application code is reinterpreted each time a metalevel operation is invoked. In contrast, our approach is restricted to customizing distributed interactions. Thus, our metaarchitecture may be implemented using compiled rather than interpreted objects. This approach leads to lower overhead imposed on the underlying application. C protocol is first installed. Other operations check the validity of role assumption and execute the binding of roles to participants.

To install a protocol, a protocol instance is created; the instance operates independently of all other protocol instances. Protocol operations may be invoked on the protocol instance to assign roles to components in the system. Roles in a protocol are relative to a particular protocol instance. Consider the example in Figure 3: in the primary-backup protocol, component B is in the role of server and A is in the role of client; in the periodic checkpoint protocol, A is in the role of server. The independence of each protocol instance is essential to keep protocol design modular: protocols may be written independently and then composed at runtime. A metalevel stack is used to separate the roles assumed by a particular component (see the previous section and Figure 3). The order in which roles are assumed determines the ordering of the metalevel stack. Roles may attach "tags" to messages so that protocol-specific state may be passed to roles assigned to other components.

Example

To implement the video-on-demand system, the media library must ensure that payment for the movie as well as issuance of a receipt to the client is atomic. The interaction policy enforcing atomic transactions is realized through an interface between the protocol implementing the policy and the application. The interface to the protocol is simple: the `beginAction` and `endAction` events delineate the beginning and end of the transaction. The protocol ensures all messages sent between the invocation of these events by the application are invoked as a single atomic action. The interface to the application from the protocol is more involved. The application must handle the following events:

`query`: The application checks an action to see if it may be performed.
`abort`: The action could not be performed atomically. The application must be able to handle such cases.

To implement this interaction policy, a two-phase commit protocol is used. Two-phase commit guarantees atomicity in spite of possible crash failures or the inability to accept an action by one of the participants in the protocol. For example, if the bank server cannot cash the check due to insufficient funds or an unauthorized signature, then the ticket will never be issued to the client.

Figure 4 gives the DIL specification of a two-phase commit protocol. The protocol is divided into two roles. The initiator role is assigned to the component, which initiates an atomic interaction (for example, by issuing `beginAction`). The server role is assigned to all other components participating in the interaction. In the video-on-demand system the media library assumes the role of initiator, while the client and bank assume the role of servers.

The initiator role supports three events: it customizes the default transmission behavior (that is, the transmit event), and handles the application-invoked events `beginAction` and `endAction`. The latter two events delineate the beginning and end of a transaction. The transmit event is invoked by the system whenever the application component sends a message. If a transaction is being constructed, the message is added to the transaction, otherwise it is transmitted immediately.

Once the `endAction` event has been received, the initiator and servers exchange messages in two phases in order to determine whether or not the transaction may be committed. In the first phase, the initiator sends a prepare message to each server asking it to vote on the current transaction. In response, a server invokes a query event on the application. Depending on the result of the query, the server either sends a `voteAbort` or `votePrepared` message to the initiator. If the initiator receives any abort messages, then the entire transaction is aborted: each server receives a `doAbort` message and invokes an abort event on the application. Otherwise, once all servers have replied, the initiator sends a commit message to each server, which causes it to deliver the messages in the transaction to the application.

When a new instance of the two-phase protocol is created, it is initialized with the communication address of the component initiating transactions. This component assumes the role of initiator. Other components are added to the protocol by the `addServer` operator, thereby assuming the server role. Note that multiple components may assume the server role within one instance of the protocol. For example, we might install the two-phase protocol using the syntax:

```
ProtocolInstance *pb =
      newProtocol Two-Phase(MediaLi-
 brary);
pb -> addServer(Client);
pb -> addServer(Bank);
```

Implementation

We have implemented our approach on the actor platform Broadway. Broadway provides C++ support for distributed actor programs including asynchronous communication, dynamic actor creation, and scheduling of actors. Basic actor functionality is augmented with support for migration, exception handling, and synchronization constraints [6]. These additional features greatly simplify the development of distributed programs. The platform currently runs on Ultrix for DEC MIPS workstations, on Solaris for Sun Sparcstations, and IRIX for SGI workstations. Each actor is implemented as a C++ object: the state and methods of an actor are the state and methods of a C++ object.

Each actor also maintains a mail queue to buffer incoming messages. When an actor is ready to process its next message, the scheduler invokes the correct method in the C++ object as a new thread. Only one method may be active for a single actor: there is no internal concurrency.

Results from experiments on SparcStation IPX workstations.

Experiment	Custom	DIL	% Overhead
Single Node	2.50 s	2.60 s	4.0%
Distributed	50.70 s	50.94 s	1.7%

Broadway supports customization of actor communication through reflection, thereby providing an implementation platform for DIL. All actors may be customized including metalevel actors. As with regular actors, metaactors are represented as C++ objects. By supporting the customization of metalevel actors, we enable the composition of protocols. Broadway supports the five system-level events required by application and meta-actors: `dispatch, deliver, transmit, reifyState,` and `reflectState.`

To implement DIL, each protocol is converted into multiple C++ classes that implement metalevel actors. One class is created for each role in the protocol as well as a manager class that controls protocol operators. Role assumption must be atomic across all participants in the protocol and support for this assumption is provided by Broadway. To avoid dependence on a single atomicity protocol, Broadway loads a user specified atomicity protocol upon initialization.

Broadway implements DIL with minimal overhead. The table appearing here shows performance measurements taken on a primary-backup protocol installed on a small database application. Each measurement represents the time required to perform 1000 interactions, where an interaction consists of receiving a message, modifying the message in memory, and sending a copy to the backup. Two versions of the application were developed: the first involving a custom-made hand-coded version of the protocol embedded into the application code and the second using DIL. Timing results were taken to compare the cost of using reflective objects to hand-coding. The results suggest that our approach may be used without excessive overhead. Note that a more computationally intensive application would lead to reduced overhead as message passing cost would play a less significant role in the performance of the application.

Discussion

Modularity and customizability are important techniques for software development. Modularity is one attraction of toolkits for distributed programming. Toolkits support a small set of protocols for interconnecting application components. Toolkits, although lacking generality, are well suited to applications requiring only the protocols they provide. For example, transaction systems are ideal for distributed database applications, but unsuitable for applications involving real-time synchronization.

Customization of component interactions has been explored in the x-Kernel [7], Maud [3], and more recently in Horus [11]. The x-Kernel and Horus utilize protocol stacks to support customization. Each layer in the stack supports a static interface for interaction with the layers above and below it. The interface in these systems is fairly elaborate. Maud supports metalevel customization of protocols for fault-tolerance. In comparison to the protocol stack approach, reflection enables Maud to use a simple but flexible interface. DIL builds on Maud; by treating protocols as single entities, it increases the granularity of abstraction at which we may develop and modify distributed systems. More recently, this work has been extended and applied to resource management policies for collections of application components [4], including Java-based support for actors and metaarchitectures (see osl.cs.uiuc.edu/foundry).

Although both the computational power of individual processors and their interconnectivity have increased by orders of magnitude, methods for developing and maintaining distributed software have not. The increasing complexity of distributed software systems in the absence of corresponding advances in software technology fuel a perennial crisis in software. Advances in sequential programming have historically come from the development of abstractions to supportreuse and separate components. We believe that methods that separate interactions from application behavior provide the corresponding ability to simplify the development and maintenance of complex distributed software. C

References

1. Acceta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A new kernel foundation for unix development. In *USENIX 1986 Summer Conference Proceedings,* (June 1986).
2. Agha, G. Concurrent object-oriented programming. *Commun. ACM 33*, 9 (Sept. 1990), 125–141.
3. Agha, G., Frolund, S., Panwar, R. and Sturman, D.C. A linguistic framework for dynamic composition of dependability protocols. In Volume VIII of *Dependable Computing and Fault-Tolerant Systems.* Elsevier, 1993.
4. Astley, M. and Agha, G. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Proceedings of Sixth International Symposium on the Foundations of Software Engineering (SIGSOFT '98)*, November 1998.
5. Campbell, R., Islam, N., Raila, D., and Madany, P. Designing and implementing choices: An object-oriented system in C++. *Commun. ACM 36*, 9 (Sept. 1993), 117–126.
6. Frolund, S. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization.* MIT Press, 1996.
7. Hutchinson, N.C. and Peterson, L.L. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering 17*, 1 (Jan. 1991), 64–75.
8. Jamali, Thati, P., and Agha, G. An Actor-based architecture for customizing and controlling agent ensembles. *IEEE Intelligent Systems 14*, 2 (Apr. 1999), 38–44.
9. Ren, S., Agha, G.A. and Saito, M. A modular approach for programming distributed real-time systems. *Journal of Parallel and Distributed Computing, 36* (1996), 4–12.
10. Smith, B.C. *Reflection and Semantics in a Procedural Language.* Technical Report 272, Massachusetts Institute of Technology. Laboratory for Computer Science, 1982.
11. van Renesse, R., Birman, K.P., Friedman, R., Hayden, M. and Karr, D.A. A framework for protocol composition in horus. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, August 1995.
12. Zarros, A. and Issarni, V. A framework for systematic synthesis of transactional middleware. In *Proceedings of Middleware'98* (Sept. 1998), 257–272.

Mark Astley (mastley@us.ibm.com) is a research staff member at IBM T.J. Watson Research Center.

Daniel Sturman (sturman@watson.ibm.com) is Manager of Distributed Messaging Systems at IBM T.J. Watson Research Center.

Gul Agha (agha@cs.uiuc.edu) is Director of the Open Systems Laboratory and Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Department of Computing and Information Science
Department of Mathematics and Statistics

Canada Research Chair in Biocomputing, Biomathematics or Biostatistics

The University of Guelph seeks applications or nominations to fill a Tier I or Tier II Canada Research Chair (CRC) position in an area of biocomputing, biomathematics or biostatistics. The appointment may be in the Department of Mathematics and Statistics or in the Department of Computing and Information Science; a joint appointment is also possible. The Departments together have 41 faculty, offering strong teaching and research programs to the Ph.D. degree. Research programs are enhanced by SHARC-Net, a 500 node Beowulf supercomputer cluster. Further information concerning the departments is available at **http://www.mathstat.uoguelph.ca** and **http://www.cis.uoguelph.ca**. The candidate's research should complement the University of Guelph Strategic Research Plan, which can be found at **www.uoguelph.ca/Research/programs/crc/3a.html**. Further information on the CRC Program may be found at the program website at **http://www.chairs.gc.ca**.

The successful candidate will be expected to maintain a vigorous research program, develop a strong graduate program and teach at both the graduate and undergraduate levels. Applications or nominations should include a curriculum vitae, a five-year research plan and four references, at least one of which addresses teaching. Forward applications to: **Dr. O.B. Allen, Department of Mathematics and Statistics, University of Guelph, Guelph, ON Canada N1G 2W1 (fax: 519-837-0221); or to Dr. W. Dobosiewicz, Department of Computing and Information Science (fax: 519-837-0323).** Applications should be received by **September 1, 2001**. All CRC appointments are subject to review and final approval by the CRC Secretariat in Ottawa.

The University of Guelph is committed to an employment equity program that includes special measures to achieve diversity among its faculty and staff. We therefore particularly encourage applications from qualified aboriginal Canadians, persons with disabilities, members of visible minorities and women.

University of Guelph

Governors State University

Computer Science Faculty

Governors State University (GSU) seeks applications and nominations for a full-time tenure-track faculty position in computer science to contribute to established B.S. and M.S. degree programs in Computer Science. We are seeking candidates with a strong commitment to teaching and with research potential. While the focus of this position will be on the practical aspects of computer programming, applicants from all areas of computer science will be considered. Duties will include teaching, research, and service. ***Qualifications:*** Earned Ph.D. in Computer Science or closely related area is preferred; a strong interest and background in programming and teaching programming languages (C, C++, VB, and Java) is required; technical experience in applications development (e.g., Windows programming and Internet programming) is required; experience working with students from diverse populations and backgrounds. The position is available September 1, 2001. The review of applications will begin immediately and will continue until the position is filled. To apply, interested candidates should send a letter of application addressing qualifications, a curriculum vitae which includes identification of the teaching/research specialties, a copy of transcripts; and the names, addresses and appropriate contact information of three personal references to: **Computer Science Search Committee, c/o Division of Science; Governors State University; University Park, IL 60466.**

Governors State University

University Park, IL 60466-0975

AA/EOE