Object-Oriented Tools for Distributed Computing

Richard M. Adler
Symbiotics, Inc.
725 Concord Avenue
Cambridge, MA 02138

ABSTRACT

Distributed computing systems are proliferating, owing to the availability of powerful, affordable microcomputers and inexpensive communication networks. A critical problem in developing such systems is getting application programs to interact with one another across a computer network. Remote interprogram connectivity is particularly challenging across heterogeneous environments, where applications run on different kinds of computers and operating systems. NetWorks!™ is an innovative software product that provides an object-oriented messaging solution to these problems. This paper describes the design and functionality of NetWorks! and illustrates how it is being used to build complex distributed applications for NASA and in the commercial sector.

INTRODUCTION

A distributed computing system consists of software programs and data resources that are distributed across independent computers connected through a communication network [1]. Distributed systems are proliferating, owing to the availability of powerful, affordable microcomputers and inexpensive communication networks. Examples include: network operating systems for sharing file and printer resources; distributed databases; automated banking and reservation networks; office automation; groupware; and operations control systems for manufacturing, power, communication, and transportation networks.

In traditional uniprocessor systems, users connect with a mainframe or minicomputer from remote terminals to share its centralized data stores, applications, and processing resources. In distributed systems, both computing resources and users are physically dispersed across computer nodes on the network. Distribution offers important advantages, including replication, fault tolerance, and parallelism. For example, heavily-used programs or data may be duplicated on multiple nodes to increase resource availability and reliability. In addition, individual applications may be distributed, enabling their constituent tasks to be executed simultaneously on dedicated computers.

These benefits are achieved at the price of increased design complexity.[1] In particular, users and programs must frequently access information resources and applications that reside on one or more remote systems. Such access presupposes solutions to the design problems of:

- establishing interprogram communication between remote applications.
- coordinating the execution of independent applications, or pieces of a single distributed application, across networked computers.

Interprogram connectivity enables applications to exchange data and commands without outside intervention (e.g., users explicitly transferring and loading files). Such capabilities are critical for highly automated distributed systems that support activities such as concurrent engineering, data management and analysis, and process or workflow control. Connectivity also enables end-users to access remote database or file system resources interactively, via graphical user interface programs.

---

[1]Obvious space limitations preclude discussion of important design issues such as: (a) locating distributed resources, which may be replicated and/or movable; (b) maintaining consistency across replicated data stores or distributed computations; and (c) managing recovery from partial failures such as node crashes or dropped network links in an orderly and predictable manner.

Coordination dictates how distributed applications interact with one another logically. Common models include client-server, peer-to-peer, and cooperative groups [2].

Three basic strategies have been developed to address the problems of distributed interprogram communication and control — network programming, remote procedure calls, and messaging systems. This paper focuses on NetWorks!™, a novel messaging system that is based on advanced object-oriented software technologies. NetWorks! provides a network computing solution that is generic, modular, highly reusable, and uniform across different hardware and software platforms. This last attribute is important because many distributed computing tools are restricted to homogeneous environments, such as PCs or workstations. NetWorks! conceals the complexities of networking across incompatible operating systems and platform specific network interfaces, enabling application developers that lack systems programming expertise to build complex distributed systems.

The next three sections of this paper review and compare network programming, remote procedure calls, and conventional messaging systems. The fourth and fifth sections describe the NetWorks! system and illustrate its use through distributed applications in the domains of process control and office automation. The sixth section discusses extensions to NetWorks! that are currently being commercialized, which highlight the benefits of its innovative object-oriented approach.

NETWORK PROGRAMMING

Networked computers exchange data based on a common set of hardware and software interface standards, called protocols [3]. Examples include Ethernet, TCP/IP, LU6.2, and DECNet. The low level physical protocols consist of devices resident in computers connected together with a cable (e.g., Ethernet interface cards and thinnet cable). Programs called device drivers enable a computer's applications and operating system to communicate with its physical interface to the network. Higher level software protocols dictate how remote systems interact during the various phases of data exchange. For example, the receiving computer may need to acknowledge receipt of data to the sender. Both systems must agree on the convention (i.e., datum) used to signal acknowledgments.

Software network protocols have an advertised application programming interface (API), which takes the form of a library of function calls. Network programming consists of using protocol API libraries to define interactions between remote applications. Specifically, developers must insert API calls into programs to: initiate network connections between source and target programs across their respective host computers; send and receive the required data; and terminate connections. For example, developers might use network programming to connect diverse tools for computer-aided engineering. This would enable users to extract a design model from a database on one computer, apply a simulation program to the model on a second, analyze the results, and revise the design with another tool on a third node.

Network programming is complex and highly detailed: every phase of an interaction must be handled explicitly (e.g., managing network connections, the sequence of data exchanges that constitutes a complete "session," translating across the data and command interfaces of different applications). Network programming is particularly difficult across heterogeneous computers, requiring specialized expertise with incompatible protocols and operating systems. Typically, the code required to tie programs together is highly application-specific, which limits opportunities for reuse. Moreover, network programming code is generally tightly coupled to application-specific behaviors, impeding maintainability and extensibility of both the applications and their distributed computing interfaces.

REMOTE PROCEDURE CALLS

Remote Procedure Calls (RPCs) represent a second generation of tools for distributed computing [4]. RPCs are commonly used to implement client-server models for distributed interactions. One program, called a client, requests a service, which is supported by some other application, called a server. Upon receiving a

client request, the server responds by providing the requested service. Common examples of client-server computing based on RPCs include database, network file, and network directory services.

RPCs extend the familiar function call mechanism used on single computers to work over the network. In the single processor case, a program invokes a subroutine and blocks until the subroutine returns results and control. RPCs distribute this model by using special programs called "stubs" to link applications that reside on different computers. An RPC from an application is automatically directed to the relevant (local) client stub program. The client stub dispatches the call to the corresponding (remote) server stub, which invokes the target application. The latter eventually completes its processing and transfers results and control back to the server stub. This stub relays the results back to the client stub, which returns them to the original calling application, as depicted in Figure 1.

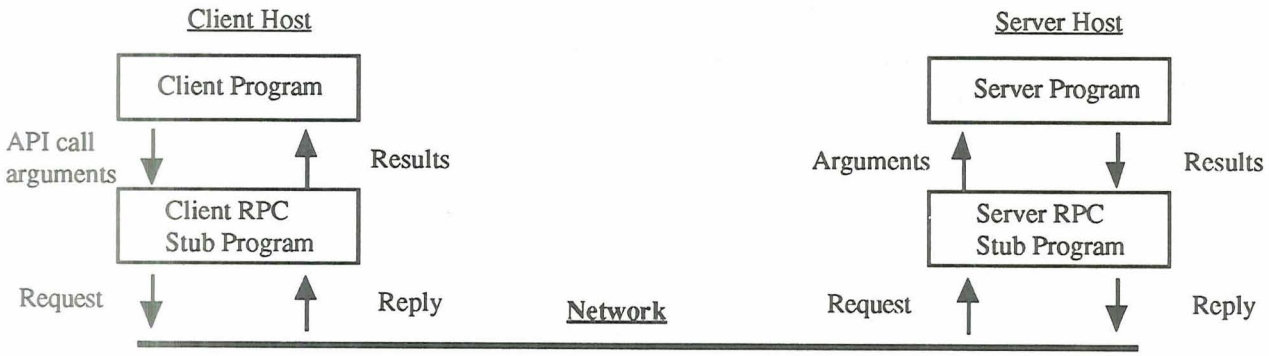

Figure 1. Client-Server Interaction using Remote Procedure Call Model

RPC tools generally include an API call library, a specification language, and a stub compiler. Developers insert the necessary RPC API calls into the source and target application programs. They then use the specification language to define the desired aspects of the interaction between the remote applications, such as packing and unpacking call parameter values and results, and managing network connections for transporting request and response data. Developers then compile their specifications to generate the RPC stub programs.

RPC tools represent a major improvement over programming with network protocols. First, application developers are already familiar with traditional, nondistributed procedure call models. Second, distributed behaviors are largely isolated within RPC stub programs, fostering modularity and maintainability. Third, stub compilers automatically generate code for interfacing with low-level network protocols. This greatly reduces the amount of programming required to define interactions between remote applications as well as the amount of systems level expertise required to design a distributed system. Fourth, RPCs specify program interfaces in terms of named operations with type signatures for call parameters. This means that RPC calls can be debugged for argument typing errors at compile time. Finally, RPC implementations are widely available and are often bundled into operating systems (e.g., Unix V4.2 BSD, the Open Software Foundation's Distributed Computing Environment).

RPC tools also have important limitations. First, RPCs are not fully compatible across disparate operating systems, network protocols, and tool vendors, although standards are emerging. Second, any distributed behaviors that cannot be expressed via the RPC specification language must be implemented using network programming. Intermingling custom code with the stub programs generated by the specification compiler complicates maintainability and extensibility. Third, the specifications for RPC stub programs are not readily reusable across applications. Fourth, every RPC-based service must be initialized explicitly and must always be executing, which wastes processing resources. Finally, the simple function call mechanism underlying RPCs is difficult to extend from client-server architectures to more advanced interaction models. In particular, standard RPCs are blocking or synchronous: such behavior is inefficient as it precludes calling programs from performing other tasks pending return of

results. RPCs are also inherently pairwise and unidirectional, making them unsuitable for distributed control models such as peer-to-peer, extended conversations, and cooperative groups.[2]

MESSAGE-PASSING MODELS

Message-passing models provide more flexibility for constructing distributed interactions than function call semantics. In particular, the act of passing a message need not commit the sending program to block pending replies. For example, a Request-Only model sends a single message from a source to a target process. It is used primarily for efficient, "side-effect" interactions, such as distributing information from data feeds. Similarly asynchronous (i.e., non-blocking) Request-Reply models support concurrent computing, in which programs dispatch service requests and immediately proceed with other activities pending the return of replies. The added reply message can be used to return either results of remote computations or acknowledgments that requested actions such as database updates have been completed. RPCs correspond to the synchronous Request-Reply model. Other messaging models include broadcasting (one-to-all), multicasting (one-to-many), and cooperative groups, which encompass specialized control protocols such as voting or negotiation. Note that the flexibility afforded by asynchronous models incurs distributed control overheads for tracking pending messages and responses.

Messaging systems come in two basic varieties, pipes and object-oriented. A pipe establishes a stream or channel between two applications [5]. Typically, pipe tools supply an API library for initiating pipe connections, writing to them, checking status, and reading from them. For example, data may be sent down a pipe from a source process and received at a sink process. All of the network level programming required to create pipes and transport messages through them is built into the pipes tool, and concealed from developers through the high level API. Pipes tools generally use the asynchronous Request-Only model, which is useful for efficient point-to-point transfer of bulk data.

Queue-based messaging represents a variation on pipes that establish connections between computers rather than specific programs. Such tools typically consist of two components, an API library and a queue management system. A client-server application would be implemented as follows (cf. Figure 2). First, the client program executes a Send API call, which posts a request message to the target server onto the local outbound queue. The local queue manager dispatches the message to the target node, where the remote queue manager posts it to an inbound queue. The server program uses a Receive API call to retrieve the request from the inbound queue, and then performs the required processing. It then uses a Send call, which posts the response to the outgoing queue. The remote queue manager sends this reply message back to the inbound queue on the client node, where the client program can retrieve it. Recently, directories have been incorporated into some messaging systems, which identify the nodes where registered services are located. This design feature enables client programs to issue service requests transparently, without having to know in advance or specify the server's whereabouts.

Message-based tools are much simpler to use than RPCs in that they dispense with stub specifications, compilers, and custom network programming. The messaging API calls completely conceal interfaces to network protocols, which fosters uniformity and portability of messaging tools across heterogeneous systems. However, conventional messaging systems fail to partition code for distributed computing in a manner analogous to RPC stubs. The logic for packing and unpacking message contents and for polling the pipe or queue to check message status is coded into the application program proximate to messaging API calls. Moreover, models for distributed interaction that are not directly supported (e.g., cooperative groups, multicasting, reliable transaction protocols) must be implemented using the available API library functions. A substantial amount of distributed computing code may need to be inserted into applications to realize the requisite logic, data management, and interprogram coordination. Although developers could apply structured design techniques, a more elegant strategy is to promote modularity

[2]A few RPC tools have been extended to try to address these objections (e.g., with callbacks or futures); however, such modifications violate the basic semantics of procedure calls.

within the framework of the distributed computing tool itself. Ideally, the tool would also foster reusability of distributed functionality, another feature that is lacking in conventional messaging tools.

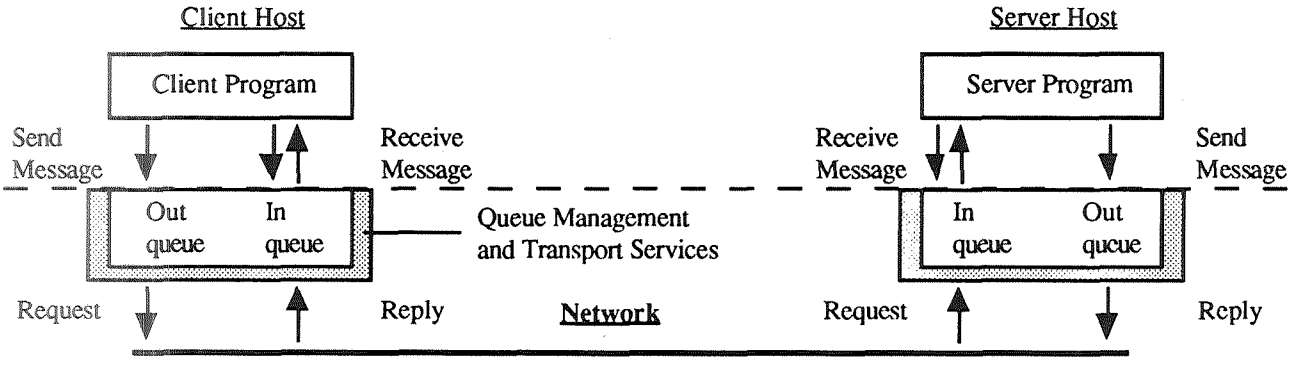

Figure 2. Client-Server Interaction using a Queue-Based Messaging Model

NETWORKS — OBJECT-ORIENTED MESSAGING

Like conventional messaging systems, NetWorks! defines a uniform high level communications interface between distributed programs running on heterogeneous platforms. In particular, a NetWorks! Messaging Facility (NMF) resides on all nodes of the network. The NMF provides queues, queue management, and message transport facilities. The NetWorks! architecture differs from conventional messaging systems in two primary respects. First, it introduces objects called Agents. Agents provide the locus for: (a) the control logic for passing and retrieving messages; (b) packing and unpacking application data for messages; and (c) distributed coordination. Intrusions into applications are thereby limited to high-level API calls, such as telling Agents to initiate messages for distributed interactions. Thus, instead of manipulating the NMF message queues directly, applications interact with Agents. In this respect, Agents play a role analogous to RPC stub programs. A basic NetWorks! model for a client-server interaction is depicted in Figure 3. More detailed examples are presented in the next section.

Second, the NMF incorporates a scheduler, which automatically manages incoming messages. Basically, the scheduler identifies the target Agent from each message and initiates the execution of that Agent for the given message. In essence, the scheduler delivers incoming messages to the relevant Agents, which then interact with their associated applications by injecting appropriate data or commands. This architecture greatly simplifies the control of asynchronous interactions.[3] The NMF scheduler is basically nondeterministic, although the messaging API enables developers to specify values for a priority attribute to control precedence ordering of messages explicitly.

NetWorks! provides a development environment for defining Agent objects and for exercising and debugging their behavior interactively. An Agent consists of: (a) conventional program code, such as C or C++, (b) calls to the Agent API library for interacting with the NMF; and (c) calls to the native API of the Agent's associated application(s). Agents can be created on one platform and generated and installed (i.e., compiled and linked) remotely, across heterogeneous computers, for true distributed development. Developers then connect applications to Agents using a high-level messaging API, which is uniform across different programming languages. Agents process API calls from applications to prepare messages, which are posted to the NMF for transport. Agents also receive incoming messages and inject them directly into their associated applications, simplifying control for asynchronous interactions.

---

[3]For flexibility, the NetWorks! messaging API supports function calls to retrieve messages from Agents in situations where polling behavior is desired.

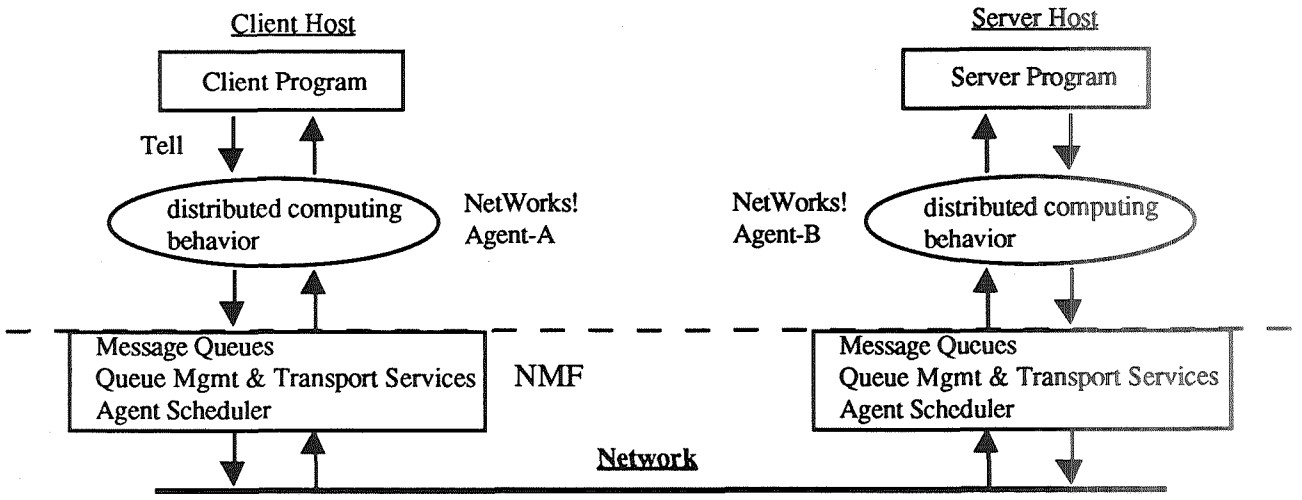

Figure 3. Client-Server Interaction using NetWorks! Object-Oriented Messaging Model

NETWORKS! APPLICATION EXAMPLES

Process Control

NetWorks! Agents support an object-oriented approach to remote interprogram connectivity. Objects are data structures that contain state information and behavior, which consists of operations for accessing and manipulating state information [6]. These operations, called methods, are invoked by sending messages to particular objects. This message-passing model maps directly into distributed interactions among application elements viewed as objects. Objects can be reused by creating new subclasses, which physically reuse or *inherit* behaviors from their parent classes. Moreover, inherited behaviors can be customized (i.e., overridden) selectively, through a process known as specialization. Object models encapsulate internal state and behavior by restricting any access other than through the message-based interface. This form of information-hiding or modularity enables system elements to be developed and tested independently, for later integration.

Consider a simple distributed system that uses NetWorks! Agents to support polling of remote sensors by an application that performs fault detection, isolation, and recovery functions (FDIR). FDIR functions are critical for automating operations support for mission control centers, manufacturing, or other complex network systems. The FDIR program initiates the polling sequence (cf. Figure 4) by issuing the NetWorks! application API call:

(Tell :agent FDIR-1 "poll measurement-Z")

The message contents in this case consists of the polling command for *measurement-Z*. The *:agent* keyword indicates that this message is to be delivered to the Agent *FDIR-1*, which is assumed by default to be co-resident with Program-A. The NetWorks! *Tell* API function is asynchronous, enabling the FDIR program to move on immediately to poll other sensors or perform FDIR reasoning. Agent FDIR-1 may also support other messages from the FDIR program (e.g. for querying remote databases). The synchronous *Tell-and-Block* API function is provided to support blocking control models as well.

NetWorks! Agent objects contain two methods that control the processing of messages, called in-filters and out-filters. An in-filter parses incoming messages and then (a) invokes the Agent's associated application and/or (b) relays the message, which it may modify, to another Agent. Developers use the NetWorks! *Pass* API call to send messages from *within* an Agent in-filter method to another Agent.

(Pass :agent s-monitor :host host-2 "poll measurement-Z")

In this case, the *Pass* API function relays the polling request from Agent *FDIR-1* to the Agent *s-monitor*, which is located on the remote platform *host-2*. The *Pass* function posts the message to the local NMF, which transports it to the NMF on *host-2*. This NMF executes Agent *s-monitor*'s in-filter method, which parses the polling command message and interacts with the controller program for *sensor-Z* to collect the relevant measurement. Upon completing this process, the in-filter uses a *Set-Contents* API command to store the acquired data value in the *host-2* NMF queue and terminates. The *host-2* NMF then invokes the out-filter method for Agent *s-monitor*. Out-filters enable developers to postprocess results from in-filters. For example, a timestamp could be appended to the measurement value. The two NetWorks! NMFs automatically transport the final response message across the network to Agent *FDIR-1*, which injects it into the FDIR program using the latter's native API.

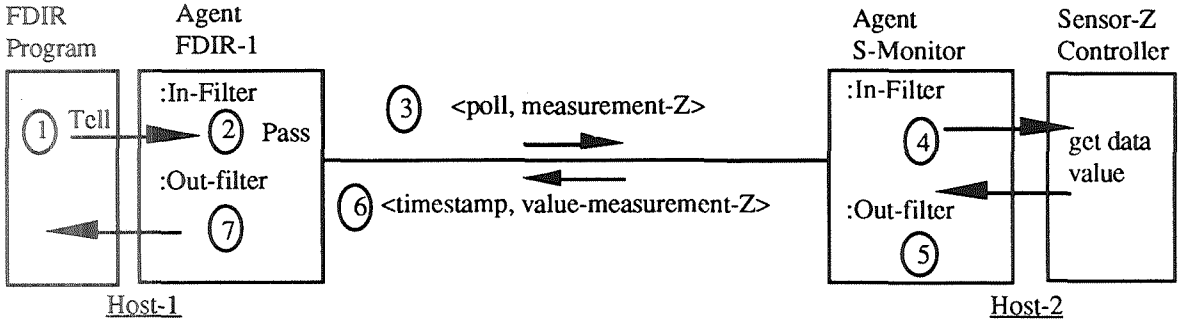

Figure 4. NetWorks! Agent-Based Message passing Model

NetWorks! Connect — DDE Over the Network

Connect is a commercial end-user software product that Symbiotics constructed using the NetWorks! development tool. Connect is a distributed application that extends Dynamic Data Exchange (DDE) over the network. DDE is a protocol within Microsoft's PC Windows operating environment that links data across two applications running on the same computer. These links are established using standard Windows application menu operations such as Copy (to the Clipboard) and Paste Link (from the Clipboard). The Paste Link operation links the copied data dynamically, so that updates made to the original source information (e.g., cells in a spreadsheet), are automatically propagated to update the linked information in some other application (e.g., a table in a word processing document).

NetWorks! Connect extends these operations to link data across DDE-compliant applications that run on different PCs across a network. Connect relies on NetWorks! Agent that copy remote Clipboards and handle DDE messages on each platform. The only departure from the single platform linking process is that users interact with a NetWorks! Windows-style menu to select a remote computer and retrieve the contents of its Clipboard. The DDE Agents transparently maintain the link produced by the Paste Link operation over the network. Connect also provides utility Agents for sending and receiving files across the network, and for exchanging *ad hoc* memos (e.g., for on line conferencing). In addition, "power" users can embed NetWorks! API calls into custom macros, Visual Basic programs, or programs in other languages that support DDE API calls. Lastly, because NetWorks! runs on heterogeneous platforms, such Connect applications can exchange data with programs on non-Windows platforms.

EXTENSIONS TO NETWORKS! — THE AGENT LIBRARY

An important advantage of NetWorks! over conventional distributed computing tools is that it promotes reusability through object-oriented inheritance of Agent behaviors. To exploit this advantage fully, Symbiotics is developing a library of Agent classes that establish predefined models for integration (Gateways) and distributed control (Managers). This library will facilitate a highly intuitive building block approach to designing and implementing distributed systems: developers can

simply select suitable library Agents and specialize them to satisfy application specific requirements. The Agent library will be commercialized next year as a layered NetWorks! product.

Gateway Agents

The Gateway Agent class defines a uniform peer-to-peer interaction model for integrating heterogeneous programs. Peer-to-peer simply means that any application Agent can act as a client or as a server with respect to any other. This control model, which is implemented via in-filter and out-filter methods, invokes a set of auxiliary Agent methods in a data-driven manner to process:

- API calls from the Gateway's application to initiate service requests (client behavior).
- incoming request messages from other application Gateway Agents (server behavior).
- responses to prior outgoing service requests from the Gateway (client behavior).

These messages all conform to a standard format, which enables the Gateway control model to determine which type of behavior is required.

An application is integrated into a distributed system by creating a new Gateway subclass and specializing two sets of Agent methods. One set, called translator methods, maps information across incompatible data and command interfaces for independent applications. NetWorks! incorporates a Data Management Subsystem (DMS), which supports a uniform "neutral exchange" format for all Gateway messages. DMS defines an object-based model for representing both primitive data types (e.g., character, integer, float) and composite types (e.g., database records, frames, arrays, files). A high-level API enables developers to create new composite types, and to create, pack, and unpack instances of these primitive and composite types. (Note: RPCs often offer similar tools, such as XDR [7].) A Gateway Agent's translator methods use the DMS API along with the application's API to map between the neutral exchange and native formats. The second set of Gateway methods defines the program's desired client and server behaviors, using the translator methods as a high level API to move data and commands into and out of the application non-intrusively. These methods typically consist of program Case statements, whose individual clauses dictate specific client or server behaviors. Gateways thereby separate communication from data management, and cleanly partition both kinds of functionality from the behaviors required for an application to participate in a distributed interaction.

Gateway Agents promote reusability of code in two ways. First, every application Gateway subclass inherits the uniform peer-to-peer control model from the root Gateway Agent class. Gateways thereby conceal and reuse common message passing *and* control behaviors for inter-Agent communication. Second, applications are often implemented using a development tool, such as 4GLs, CASE products or AI shells. In these situations, translator methods can be defined once, in the tool-specific Gateway subclass. Subclasses of that Gateway then inherit both messaging and information mapping functionality, leaving the developer to specify only the desired client and server behaviors for particular applications. The uniformity and modularity afforded by Gateways promote maintainability and extensibility, which are particularly important to support large, complex distributed systems that evolve over extended lifecycles in the field.

Manager Agents

Gateway Agents integrate distributed applications, facilitating basic peer-to-peer interactions. Manager Agents define reusable control models for coordinating more complex interactions. For example, a Hierarchical Distributed Control (HDC) model decouples application Agents from direct connections with one another [8]. Application Gateways send request messages to the HDC-Manager, which posts them to a task agenda (cf. Figure 5). The HDC-Manager contains a directory that identifies the application Gateway, its location, and its request interface for each service that is available in the distributed system. Using this directory, the HDC-Manager processes its agenda asynchronously, dispatching requests to the supporting Gateway for each pending task. Each such Gateway posts service

results back to the HDC-Manager, which relays them directly back to the relevant requesting Gateway Agents. Using this model, application Gateways need only know (a) how to interact with the HDC-Manager, and (b) the services that are available through it — no prior knowledge is needed about the identity, location or access interface of any other Agents. This design also promotes maintainability and extensibility for complex distributed systems: the HDC-Manager directory creates a layer of control abstraction that insulates application Gateways from modifications to each other and additions of new application Gateways and services.

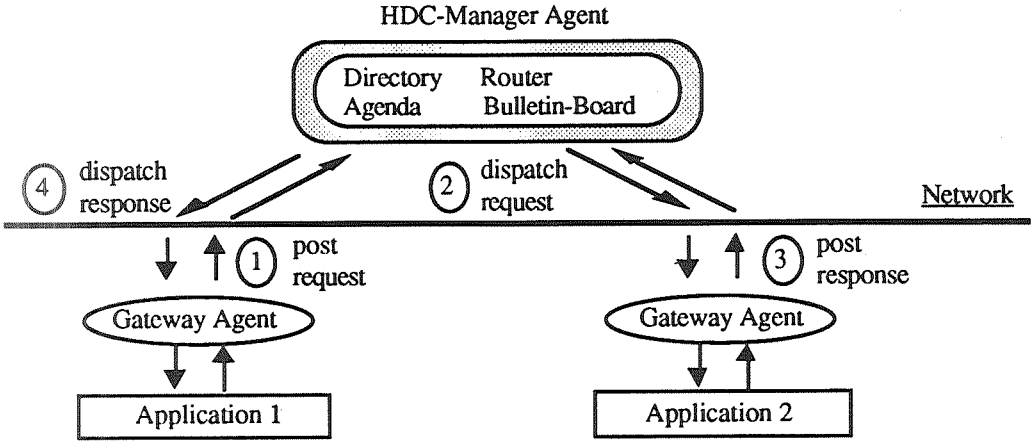

Figure 5. Operational model for the HDC-Manager Agent

The HDC-Manager constitutes an intelligent router that "brokers" discrete client-server interactions. However, tasks such as analyzing scientific data often require complex sequences of interactions to transport and manipulate data across distributed compute servers (e.g., signal processing and visualization tools). Defining such sequences across the relevant Agents can be difficult, particularly when those Agents participate in multiple task sequences. The Process Planner Manager supports a process-oriented distributed coordination model to solve this problem [9]. Interaction sequences are captured in process *scripts*. The Process-Planner coordinates the execution of such scripts by dispatching individual script elements as service requests to the HDC-Manager, acting as a driver to the HDC-Manager's router. Scripts promote maintainability and extensibility by centralizing the specification of sequences of distributed interactions. Moreover, the ability to combine Managers into hybrid control architectures illustrates the power of the Agent library's building block approach to designing distributed systems. A third Manager Agent, called a Server-Group, supports one-to-many client-server computing. This Agent decomposes complex requests into simpler services that independent server Agents can process concurrently. The Server-Group then collects and processes the results, assembling them into a single response. This interaction model is useful for groupware and decision support systems (e.g., collecting information dispersed across many databases and performing relational joins or analysis).

Symbiotics is working with NASA to refine and apply the NetWorks! Agent library to build several complex distributed systems. For example, NASA has developed various expert systems that "retrofit" the Launch Processing System for the Space Shuttle with automated FDIR capabilities. NetWorks! Gateway and HDC-Manager Agents are being utilized to integrate and coordinate these independent systems to work together cooperatively (e.g., by sharing data and diagnostic reasoning). A second NASA project is using Gateways to integrate tools for planning and scheduling ground operations for Shuttle missions. HDC-Manager and Process-Planner Agents are being used for script-based control of complex distributed decision support activities. Other contracts have investigated using library Agents: to design a framework for developing and managing software projects across heterogeneous computing platforms; to integrate tools for computer-aided design; and to develop group-based models for exploiting redundant applications to enhance reliability and corroborate problem solutions.

CONCLUSIONS

A central problem in developing distributed computing systems is enabling applications to interact across the network. Solutions to this problem are now being called "middleware," which encompasses all of the software between end-user applications and their networked host platforms. Other forms of middleware, such as network programming, RPCs, and conventional messaging tools, have one or more critical drawbacks, such as limited portability across heterogeneous platforms, intrusiveness, and limited reusability. NetWorks! middleware exploits object-oriented technologies for an advanced messaging solution that promotes modularity, maintainability, extensibility, and reusability. These characteristics are critically important to minimize development costs and to support lifecycle engineering of complex distributed systems.

NetWorks! provides non-intrusive peer-to-peer interaction models, which foster open, "plug and go" distributed computing architectures. Every application represents a potential server resource to other programs. The NetWorks! Agent library encourages an intuitive "building block" approach to developing distributed systems, in which complex distributed interactions are facilitated through service directories and simple process scripts. Complex distributed coordination functions are inherited from predefined Agent classes and customized to fit application requirements through high level APIs. NetWorks! Agents shield developers and end-users of distributed applications not only from network protocols, but also from data management functionality and complex control logic for handling messages. This technology thereby addresses the critical problems of accessing and coordinating data and computing resources in complex distributed systems.

ACKNOWLEDGMENTS

NetWorks! technologies have been developed with funding from the Small Business Innovative Research Program by the U.S. Army (contract DAAB10-87-C-0053) and NASA (contracts NAS10-11606, NAS10-11882, NAS5-31920, and NAS8-39343).

REFERENCES

[1] G. Coulouris and J. Dollimore, *Distributed Systems: Concepts and Design*, Addison-Wesley, Reading, Massachusetts, 1988.

[2] G. Andrews, "Paradigms for Process Interaction in Distributed Programs," *ACM Computing Surveys*, Vol. 21, No. 1, March 1991, pp. 49-90.

[3] W. Stallings, P. Mockapetris, S. McLeod, and T. Michel, *Handbook of Computer-Communications Standards Vol. 3*, Macmillan, New York, 1988.

[4] A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, February 1984, pp. 39-59.

[5] D. Gifford and N. Glasser, "Remote Pipes and Procedures for Efficient Distributed Communication," *ACM Transactions on Computer Systems*, Vol. 6, No. 3, August 1988, pp. 258-283.

[6] B. Meyer, *Object-Oriented Software Construction*, Prentice-Hall, New York, 1988.

[7] R. Corbin, *The Art of Distributed Applications*, Springer-Verlag, New York, 1991.

[8] R.M. Adler, "A Hierarchical Distributed Control Model for Coordinating Intelligent Systems," *Telematics and Informatics*, Vol. 8, No. 4, 1991, pp. 385-402.

[9] R.M. Adler, "Coordinating Complex Problem-Solving Among Distributed Intelligent Agents," to appear in*Telematics and Informatics*, Vol. 9, No. 4, 1992.