N94- 35068

Group-Oriented Coordination Models for Distributed Client-Server Computing

Richard M. Adler and Craig S. Hughes
Symbiotics, Inc.
725 Concord Avenue
Cambridge, MA 02138

ABSTRACT

This paper describes group-oriented control models for distributed client-server interactions. These models transparently coordinate requests for services that involve multiple servers, such as queries across distributed databases. Specific capabilities include: decomposing and replicating client requests; dispatching request subtasks or copies to independent, networked servers; and combining server results into a single response for the client. The control models were implemented by combining request broker and process group technologies with an object-oriented communication middleware tool. The models are illustrated in the context of a distributed operations support application for space-based systems.

INTRODUCTION

The dominant architecture for distributed systems today is the client-server interaction model. One application, the client, requests a service from a single provider, or server, which performs the desired task and returns the result to the client (cf. Figure 1). Examples of servers include transaction database systems, specialized graphics and numeric processing engines.

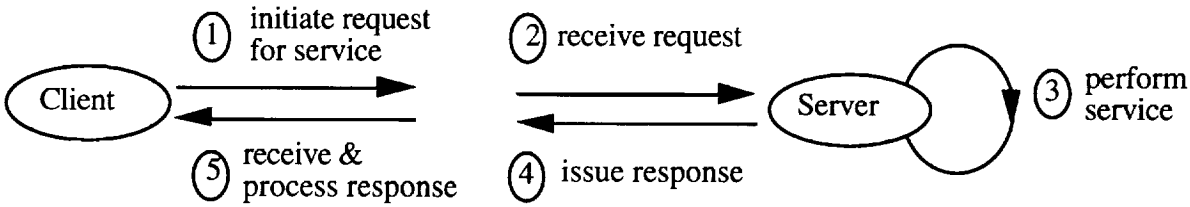

Figure 1. Client-Server model for distributed interaction

As distributed systems grow in complexity and scope, clients often need to interact with multiple servers. For example, a client may issue queries for information that is distributed across multiple, independent databases. Similarly, one application may need to notify various other programs that may be affected by its activities. Such one-to-many interactions are becoming increasingly important in domains such as decision and operations support, concurrent engineering, cooperative workgroups, office automation, and process control.

A special class of one-to-many interactions involves servers that can perform the same task(s), or *functionally redundant* systems. The most obvious form of functional redundancy is exact replication, such as distributing multiple copies of an application or database across a network of computers. Redundancy can also be achieved by replicating functionality through alternative technologies or methods. For example, intelligent advisory systems can be implemented using neural nets, rule-based or model-based reasoning systems. Combining such alternative methods exploits their complementary strengths, and can enhance the completeness, precision, and certainty of service responses.

Functional redundancy facilitates fault tolerance and resource availability, which are important attributes for mission-critical applications. Replicating servers or data resources enhances

reliability by enabling distributed systems to be reconfigured, often automatically, to recover from single point failures. Availability of scarce or heavily-used resources is augmented by allowing concurrent access to multiple distributed copies, for increased performance and convenience.

This paper describes two control models that transparently coordinate distributed interactions between a single client and multiple, possibly redundant servers. Clients access the control functions of these models through a uniform, high-level Application Programming Interface (API). The API enables clients to issue service requests and process responses more or less oblivious to the complex distributed processing that is actually required to satisfy their requests. These coordination models are implemented by combining request broker and process group technologies with an object-oriented communication middleware tool that runs across heterogeneous computing platforms.

The following sections review the basic control requirements for one-to-many client-server computing and enabling software technologies that meet these requirements. The remainder of the paper describes and illustrates the new coordination models.

COORDINATION REQUIREMENTS FOR ONE-TO-MANY INTERACTIONS

Given enabling tools for network communication, distributed control for one-to-one client-server applications is fairly straightforward. Clients initiate interactions by issuing requests. Servers respond, and clients complete the cycle by processing results. The primary source of control complexity arises from detecting errors, such as dropped network links or computer failures, and recovering from such problems in a consistent, predictable manner (Coulouris, 1988).

Introducing multiple servers complicates control requirements significantly. Three basic cases can be distinguished. A client may request a single service to be replicated across functionally redundant servers (Case 1). Alternatively, a client request may decompose into distinct service subtasks, which can be performed separately by different servers. Subtasks for such complex requests may be mutually independent (Case 2) or they may exhibit interdependencies (Case 3). Case 3 dependencies generally entail sequencing or synchronization constraints on how subtasks are performed, and are common in process-oriented or workflow applications. Adler (Adler, 1992a) describes a distributed control model that addresses this class of client-server interactions. This paper focuses on coordination models to support replicated requests (Case 1) and complex requests comprised of independent subtasks (Case 2).

Replication entails distributing a client's request to functionally redundant servers.[1] In contrast, the servers for subtasks obtained by decomposing complex service requests tend to be functionally related but disjoint. In both cases, results from servers must be collected, possibly post-processed, and returned to the client. Service results are post-processed through combinational techniques such as collation, competition, or synthesis. Collation simply collects subtask results into a consistent, uniform format. For example, status codes returned from an update operation on replicated databases might be collected into a list. Competition compares collated results with respect to one or more metrics, selectively filtering results based on their scores. A simple example is the precedence metric, or race competition, which scores results based on their order of arrival. Finally, synthesis combines and reconciles partial results. Examples include: relational join, logical union and intersection operations; voting schemes; merging and reconciling partial segments into global plans or schedules; and rank ordering

[1] For present purposes, replicated requests are assumed to be "simple" in the sense that each server responds to a copy of the request as one discrete action. Also, redundant servers incur additional requirements to coordinate the order in which they process replicated requests if it is necessary to maintain consistent state across interactions, as in mirrored transaction databases.

diagnostic hypotheses using weighted frequencies of candidates derived by complementary fault isolation methods.

The burdens of control for one-to-many interaction models can be allocated to: the client; the servers; combinations of the two; or to independent distributed coordination structures. The first three "hardwired" approaches lead to basic system engineering problems, such as limited reusability, maintainability, and extensibility across different clients and client-server associations. Accordingly, we adopted the fourth strategy, by establishing autonomous coordination models or engines. These engines, called Server Groups, function as generic, system-level servers for implementing one-to-many interactions in a distributed application. The next three sections review the distributed computing technologies that were synthesized to create two distinct Server Group engines: process groups, request brokers, and communication middleware.

Table 1. One-to-Many Client-Server Relationships

Request type	Subtasks	Server Functionality	Coordination Tasks
replicated	•	redundant	duplicate request, route requests, synchronize state, combine results, handle errors
complex	independent	distinct	decompose request, route subtasks, combine results, handle errors
complex	dependent	distinct	decompose request, sequence subtasks to reflect dependencies, route subtasks, combine results, handle errors

PROCESS GROUPS

A *process group* consists of a collection of processes, typically a set of applications, that jointly provide one or more services on a continuous basis (Liang, 1990; Kaashoek, 1993). Group processes are typically distributed across networked computers, operate in disjoint address spaces, and communicate via message-passing mechanisms. A group is *closed* if communication is restricted to members of the group; otherwise, the group is *open*. Interactions between a group and an external process (or group) are called *intergroup* communication. Interactions among members of a group constitute *intragroup* communication (cf. Figure 2).

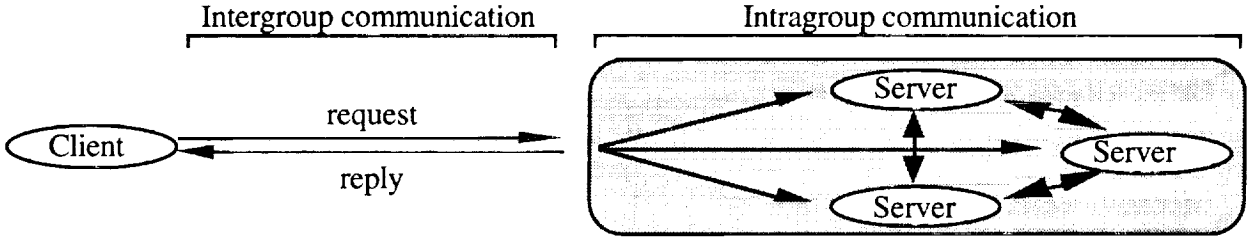

Figure 2. Open Process Group Architecture for One-to-Many Client-Server Model

Process groups provide a natural control framework for organizing collections of servers that are functionally related or redundant, mediating all intergroup and intragroup interactions between applications playing client roles and those acting as servers. The utility of process group models for supporting complex and replicated service requests hinges on the concept of *group transparency*, which simply means that a client may treat a group's service as if it were provided by a single server.

In particular, all group members are identified by a single group name, which acts as a logical address for all intergroup and intragroup communication. Clients need not track either group

membership (naming transparency), nor the host computers for individual group members (location transparency). These properties are especially attractive for interacting with groups whose membership is dynamic and/or mobile. Groups collect and post-process service results produced by member servers over extended time intervals (reply-handling transparency). Finally, clients need not deal with partial failures (fault-handling transparency), in that group interfaces either conceal single point member faults or indicate total failure. In short, group transparency fully conceals the distributed architecture and distributed behavior internal to a process group from clients.

REQUEST BROKERS

Simple, static client-server applications tend to support a limited number of distinct services, such as file, print, and database sharing. In such systems, it is relatively straightforward for client programs to keep track of the services that are available and which servers support those individual services.

In larger distributed systems, it becomes difficult to sustain this "hardwired" strategy, in which each client assumes responsibility for knowing about available services, their associated servers, and interfaces. Maintaining and extending such knowledge for individual clients is particularly cumbersome for distributed systems such as CAD or CASE frameworks, which evolve continually as server applications are enhanced, added or replaced. The notion of a service request "broker" was developed to provide clients with global, system-level support for tracking available services and servers. This strategy has been promoted by the Object Management Group, whose Common Object Request Broker Architecture (OMG/CORBA) specifies a standard for distributed object management systems (OMG, 1991).

A request broker is basically a control mechanism that mediates interactions between client applications requesting services, and server applications capable of responding to such requests. All applications that belong to a distributed system register the services that they support, their locations, and their client interfaces with the broker. Typically, the broker maintains a directory to store this information. Once this registration process is complete, any application that requires a service requests it from the broker. The broker identifies an appropriate server for the requested service using its registration directory, forwards the request, and relays back the response to the client (cf. Figure 3). (Note: OMG divides these broker and directory functions between the Object Request Broker and Object Trader Service.)

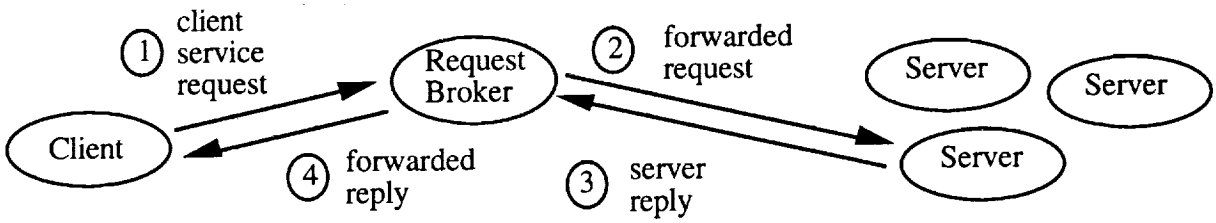

Figure 3. Broker model for forwarding client requests

The request broker architecture frees client applications from having to know where and how to obtain particular services. A client application only needs to know the names for the particular services it expects to request and how to use the request broker API to request those services. Unfortunately, current models are restricted to one-to-one interactions between a single client and a single server. The Server Group eliminates this restriction, extending the broker model to support one-to-many interactions.

OBJECT-ORIENTED COMMUNICATION MIDDLEWARE

Communication middleware refers to software tools that insulate application developers from the complexities of network programming. Middleware generally encompasses some form of systems-level software, or "kernel," together with high-level developer tools (Adler, 1992b). Kernels interface directly to the network protocols that enable communication between remote computers, such as TCP/IP and NetBIOS. Developer tools generally consist of API libraries, which direct the kernel to establish connections between specific computers and to exchange information between the desired applications. The API conceals system-level dependencies across heterogeneous computers, operating systems, and protocols.

Message-passing middleware enables applications to interact by exchanging high-level messages, as opposed to the function call approach embodied in Remote Procedure Calls (Corbin, 1991). A messaging middleware kernel consists of message queues, queue management services, and transport services (i.e. network drivers). Client applications use a messaging API to post request messages to the outbound queue of their local kernel. The kernel transparently transports the message to the inbound queue of the remote kernel, from which the remote server can retrieve it. The server replies to the client through a similar process. This model is convenient for simple client-server exchanges. However, it provides inadequate support for coordinating the complex one-to-many interactions described earlier. As a result, developers must construct suitably extended control apparatus on top of the messaging middleware, and integrate it into their applications.

The present work builds on NetWorks!, an object-oriented middleware tool that provides application connectivity across heterogeneous PC, workstation, and mainframe computing platforms. NetWorks! addresses the shortcomings of conventional messaging tools by adding a scheduler to the kernel and introducing active objects called *Agents*. An Agent consists of: (1) standard program code, such as C or C++; (2) calls to the high-level NetWorks! API library; and (3) calls to application APIs. The scheduler delivers inbound messages to the relevant Agents, which interact with local applications by injecting message data or commands. The scheduler also interacts with Agents to send messages outbound from local applications to remote kernels and Agents (cf. Figure 4). The NetWorks! API enables developers to program these messaging interactions between applications, Agents, and the kernel. NetWorks! provides an intuitive middleware implementation framework for client-server and peer-to-peer interactions because Agents can initiate and react to both request and response messages.

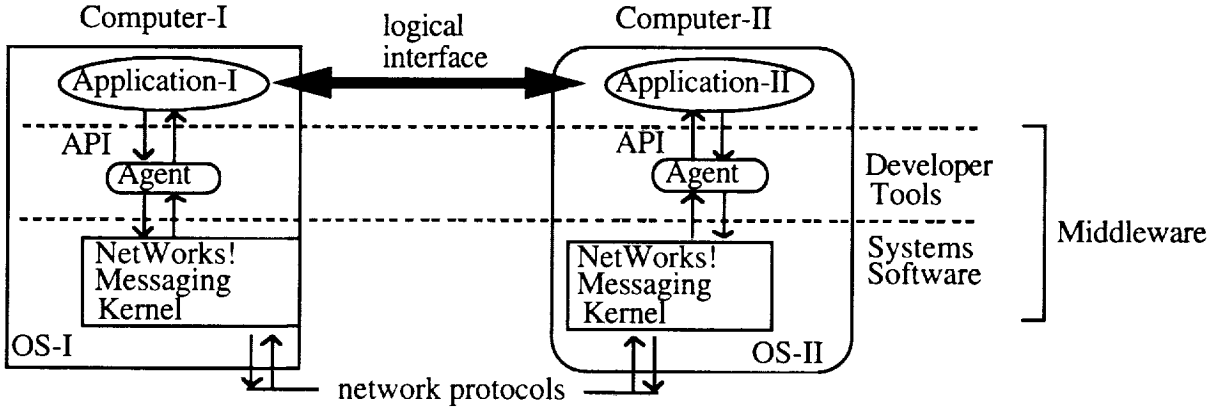

Figure 4. Conceptual Architecture of the NetWorks! middleware tool

In essence, Agents represent active participants in distributed interactions, interfacing with, but distinct from both applications and messaging kernels. As such, they constitute a separate locus

of control that can be used not only to integrate applications, but also to integrate control models for coordinating distributed application Agents. Equally important, NetWorks! Agents can reuse and selectively specialize behavior from other Agents, through object-oriented inheritance.

The NetWorks! Developer Services Library (NDSL) exploits both of these properties. The NDSL consists of a set of predefined Agents that integrate various distributed control models. Each such Agent supports a high-level, service-oriented API that conceals the underlying NetWorks! messaging API. Client applications use the NDSL API to issue service requests and retrieve responses oblivious to the Agents, messaging kernels, and complex distributed processing required to satisfy their requests. The next two sections describe the NDSL Server Group Agents, which were developed specifically to coordinate one-to-many client-server interactions.

THE SERVER GROUP COORDINATION ENGINE

The NDSL Server Group coordination model derives directly from process group technology. Client applications communicate requests to a Server Group via the high-level NDSL API. The Server Group then coordinates the activities of individual group members to provide the requested service (cf. Figure 5). The Server Group is the source of all intragroup communication for replicating or decomposing client requests, and the target of all server responses. The Server Group combines server responses as necessary, returning a single reply to the client.

The Server Group Agent exploits inheritance by reusing a parent NDSL model called the Service Request Manager (SRM). The SRM provides a conventional request broker capability for coordinating one-to-one client-server interactions. The Server Group selectively specializes inherited SRM broker object behaviors to support one-to-many client-server interactions. In particular, extensions were made to:

- the SRM API for registering and requesting services.
- the SRM broker directory, for describing functionally redundant servers, decomposing complex services, and combining results from multiple servers.
- the broker control model for dispatching requests and collecting server replies.

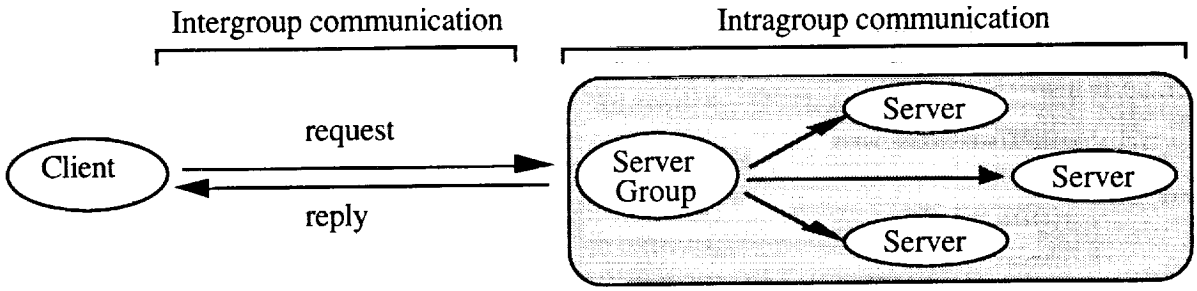

Figure 5. Server Group Architecture for functionally redundant servers

Server Group API

Standard request brokers presuppose that individual services are supported by single, unique servers. Simple directory lookup methods suffice to match client requests against registered servers. One-to-many client-server interactions require added capabilities for describing functionally redundant servers and determining which servers must be invoked to support (complex) requests. The Server Group reflects the first requirement by extending the SRM with a group-oriented registration API. Servers use this API to register as group members with a Server

Group. Membership entries coexist in the Server Group directory with service entries registered via the SRM broker API. The group-oriented API functions include:

- SGMGroupCreate — creates a group with the specified name and attributes.
- SGMGroupJoin — registers an application or Agent server.[2]
- SGMGroupLeave — removes an individual server from a group.
- SGMGroupSuspend — suspends a server from the active membership list so it is not used in current group requests, but preserves its directory entry.
- SGMGroupRejoin — cancels a server's suspension from a group, and reinstates that group member for active service.
- SGMCreateProfile — assigns server profile attribute values for a group member (used in conjunction with SGMGroupJoin).
- SGMGroupMsgSend — multicasts a message to all group members. Multicast is a selective variant of broadcasting, which automatically sends a message to all nodes specified by a membership list rather than to all network nodes.

Server Group Directory and Client Request API

The Server Group extends the SRM broker directory and API to enable multiple servers to register the same service capability. To accommodate functional redundancy, a Server Group directory entry specifies not only the service name, a server, and its location, but also a *server profile* data object. A server profile specifies that server's attributes in providing a service, such as relative speed, completeness, and precision. Server profiles are identical for servers that are exact replicas, but differ for redundant but complementary servers. Attributes take integer values, ranging from 1 (minimal) to 5 (maximal). For example, a rule-based diagnostic system might be assigned values of 3 for relative speed and 3 for completeness vs. values of 1 (slow) and 5 (very complete) for a competing model-based reasoning system. The value 0 indicates that an attribute is not relevant for a given service and should be ignored. Server profiles are extensible, allowing new attributes to be defined to meet application-specific requirements.

Client APIs for request brokers such as the SRM invoke a service by specifying the desired service name and appropriate call parameters. For example, a database query would specify "DBMS-query" as the service type and "Corporate-DB" and "select * from dept where..." as the call arguments. To support redundant servers, the Server Group client API adds a request profile object, which specifies: (1) the maximum number of servers desired; (2) the maximum difference to tolerate between server and request profiles; and (3) an embedded server profile. Thus, the server profile is used both by the Server Group to model the attributes of a server in providing a particular service, and by clients to specify the desired attributes of potential servers for carrying out that service.

Server Group Router

A conventional request broker routes a request to the server specified by the directory entry for the requested service type. The Server Group must also support replicating client requests to

---

[2] Both applications and Agents can register with an SRM or Server Group. An Agent can be viewed as an application with one exportable function (which activates the Agent). Registering an application enables clients to invoke service functions within that application. For example, a spreadsheet might register, and thereby export its recalculation function. The SRM or Server Group creates and stores handles to registered functions, which it uses thereafter to invoke the desired services for clients. The primary tradeoff is that Agents are useful in virtue of inheritance-based reusability of behaviors, whereas performance is better if application services are invoked directly rather than indirectly, via Agents.

send to functionally redundant servers, and decomposing complex service requests into requests for independent subservices. It is straightforward to extend a standard router to support request replication. However, a front-end preprocessor must be added for handling complex requests. The Server Group router algorithm: (1) decomposes the requested service if necessary; (2) extracts candidate servers from directory; (3) filters candidates with respect to an internal match algorithm; and (4) dispatches request task(s) to surviving server candidates.

The Server Group router only invokes the preprocessor (Step 1) on requests for complex services. Complex services are identified by the fact that the Server Group itself is registered as their server. To support decomposition, the server registration API requires a function pointer to be supplied for complex services. The executable code for that procedure must be co-resident with the Server Group. The procedure's input consists of the service entry from the directory corresponding to the complex service type specified in the client's request. The procedure returns a list of directory entries that correspond to the constituent subservices for that composite service.

In Step 2, the Server Group router reuses SRM lookup functionality to extract candidate servers from its directory based on the client's requested service type. In Step 3, the router executes a match algorithm, which filters redundant servers for the client's request. The match algorithm computes a differential between attribute values for the server profiles specified in the client request profile and the directory service entry. Next, it sorts server candidates in order of increasing differentials, eliminating all candidates whose differentials exceed the client's specified threshold. Finally, the algorithm returns any remaining servers up to the limit specified by the client request profile. For simple requests, the Server Group router then dispatches the given request to all surviving servers (step 4). For complex service requests, the router loops through Steps 2 through 4 for the list of subservices produced in Step 1.

This algorithm adds the absolute values of the differences between corresponding attribute values for the client request profile's embedded server profile and the directory service entry's server profile. Next, it sorts these server candidates with respect to increasing order of differential totals. It then eliminates all candidate servers whose differential totals exceed the threshold (if specified). The algorithm returns all surviving candidates up to the limit specified by the client request profile. For simple requests, the Server Group router then dispatches the given request to all surviving servers (step 4). For complex service requests, the router loops through Steps 2 through 4 for the list of subservices produced in Step 1. The overall control model is summarized graphically in Figure 6.

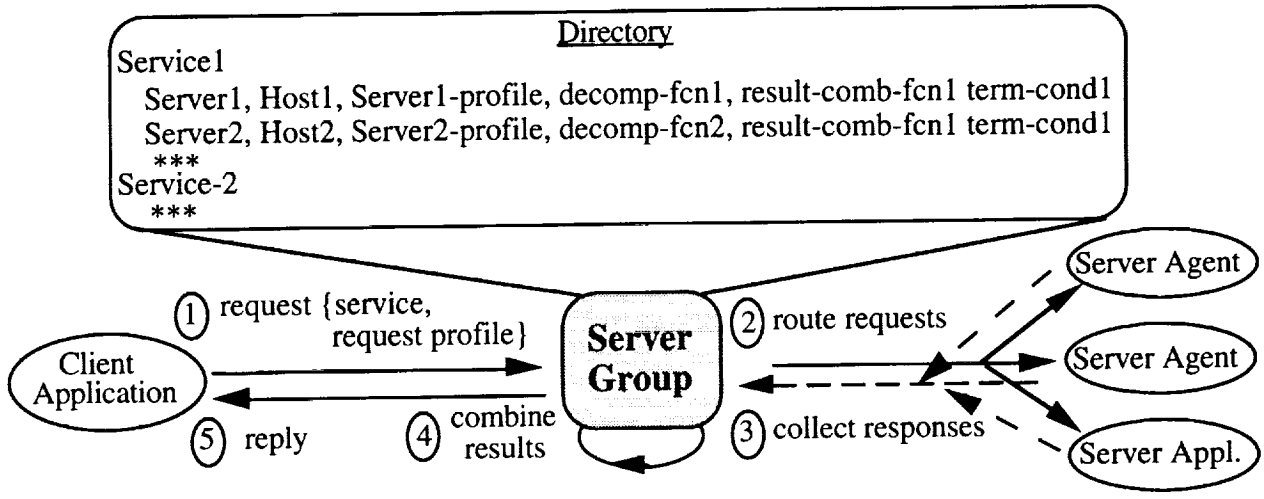

Figure 6. Overview of control sequence for Server Group coordination model

Server Group Processing of Service Results

A conventional request broker routes the single response resulting from a service request back to the requesting client. The Server Group requires more elaborate control mechanisms to collate and process responses from multiple group members. Specifically, the Server Group must know: (1) how to collect all responses for a given client request (which the Server Group may have replicated or decomposed); (2) when to stop waiting for responses for the client request; and (3) how to process the responses that have been collected up to that termination point.

With regards to issue (1), The Server Group relies on unique transaction identifiers generated by NetWorks! to label and automatically track all messages dispatched to support a given distributed interaction. With regards to (2), the Server Group uses a termination test predicate to determine when to stop collecting responses. The choice of termination predicate for a service depends largely on its desired result combination behavior. For example, all possible responses should generally be collected before applying algorithms that synthesize results. Alternatively, competitive result combination functions tend to collect the first N arrivals, or the first N arrivals that satisfy some application-specific conditions. With regards to (3), once all relevant responses are collected for a given client request, the Server Group applies the result combination procedure for the relevant service. Similar to decomposing complex services, the termination test and result combination functions are specified as pointers via the service registration API, with the caveat that the executable codes reside on the Server Group host computer.[3]

REQUESTS FOR RELIABLE, REPLICATED SERVICES

The NDSL Reliable Server Group provides a generic engine for the two phase commit protocol to support replicated databases and other transaction-oriented applications that obtain reliability through replicated servers (Ceri, 1984). The Reliable Server Group inherits most of its capabilities from the basic Server Group, specializing just two behaviors. First, the router omits the candidate filtering process; client requests are automatically multicast to *all* servers registered to support the relevant service. The Reliable Server Group then combines responses as per the standard Server Group control model. Responses indicate either success or failure (due to dropped communication links, failed host processors, or errors generated during server processing of requests). Second, the Reliable Server Group initiates another round of messages to the servers before responding to the client. Specifically, it multicasts a Commit message if all servers acknowledged success, causing them to commit their service actions as permanent transactions. Otherwise, it sends an Abort message to undo or "rollback" any temporary state changes. The two sets of messaging interactions are depicted in Figure 7. In order to exploit this reliable interaction model, server Agents or applications must be designed to support the second sequence of messages, which is absent from the standard Server Group.

[3] The Server Group filters redundant servers based on client request profiles, so it cannot determine in advance which servers will be selected. Consequently, result combination and termination test functions must be identical across all servers that register a given service.

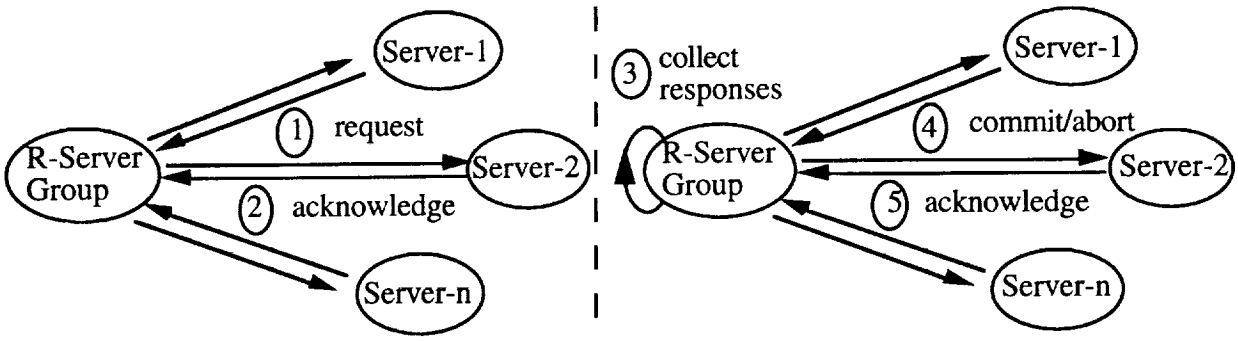

Figure 7. Two phase commit coordination model of the Reliable Server Group

DISCUSSION

Aside from the literature on process groups, research on cooperative coordination models has been most active in the context of Distributed Artificial Intelligence or DAI. Most DAI research focuses on specific types of coordination architectures for cooperating, intelligent servers, such as contract nets, blackboard architectures, and negotiation models (Bond and Gasser, 1988; Gasser and Huhns, 1989). Server Groups differ from such systems in several respects. First, Server Groups provide a centralized control module to manage passive servers, whereas DAI models tend to distribute control among autonomous servers. Second, the NDSL Server Groups were designed to support relatively coarse grained interactions among member servers, whereas DAI servers solve problems through more intensive, fine grained interactions. Third, Server Groups possess the flexibility to support radically different control behaviors for specific services simultaneously, whereas DAI models typically establish a uniform, global protocol tailored to specific (classes of) applications.

The NDLS Server Group is subject to two potential drawbacks common to centralized control models, degraded performance under heavy traffic loading, and reliability limitations due to a critical, single point of failure. The Server Group uses the NetWorks! non-blocking (asynchronous) messaging capability to minimize overheads due to group-based communication and server processing. Request decomposition tends to have minimal impact on performance except in systems with intensive request traffic and/or real-time constraints. The most serious problem arises from result combination behaviors that are computationally intensive, such as relational joins on large data sets or detecting and resolving conflicts across partial plan or schedule segments. Such processing can hold up or block brokering of lower overhead client requests. (It must be noted that decentralized models incur different, but comparable overheads in providing equivalent distributed communication and coordination functions.)

Performance can be improved significantly by isolating processing-intensive result combination behaviors, using distributed architectures that integrate SRMs with Server Groups (cf. Figure 8). All client requests are addressed to an SRM, which brokers non-group tasks and routes requests for group services involving compute-intensive processing to a remote, dedicated Server Group. The Server Group acts as a single logical server with respect to the SRM, isolating computation and concealing the group-based processing. Multiple Server Groups can be introduced as appropriate. Such hybrid configurations illustrate the power of the NDSL building block approach to combine the design simplicity of centralized coordination models with the concurrent processing advantages of distributed control architectures.

As for fault tolerance, the SRM and Server Groups are currently being enhanced for greater reliability via an automated recovery design based on standard checkpointing and message-logging techniques (Strom, 1985). Control extensions to support fault tolerant transparency are

also being investigated, involving automated detection and management of group member server failures (Birman, 1993).

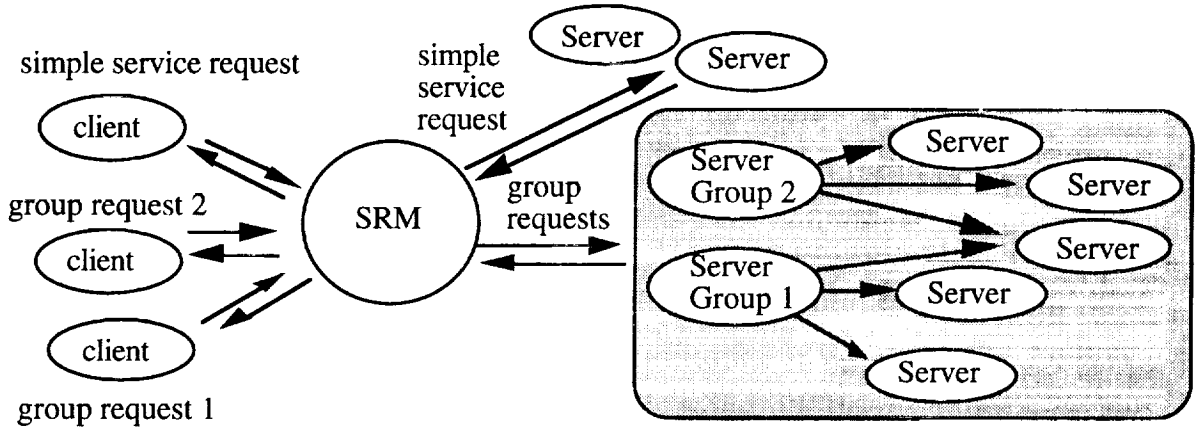

Figure 8. Hybrid Distributed Coordination Model

EXAMPLE APPLICATION

A prototype NDSL Server Group was built to simulate group-based coordination of rule-based systems for distributed operations support of the space station (Ringer, 1991; Walls, 1990). These applications automate fault detection, isolation, and recovery functions (FDIR) for the primary power generation and distribution subsystem (PGDS) and module power subsystems (MPSs). PGDS manages the supply of power to MPSs, which manage power consuming components within Laboratory and Habitation modules. The FDIR expert systems must interact cooperatively to reflect the architectural interfaces between their target systems. The FDIR Server Group coordinates these interactions. Two additional FDIR systems that do not actually exist, a model-based reasoner and a neural net, were simulated to introduce functional redundancy of FDIR services for an MPS (cf. Figure 9).

The Server Group API was used to register the FDIR servers for the PGDS and MPS and their respective services. A complex service for system-wide FDIR was also registered, which decomposes into diagnostic service requests to the rule-based FDIR systems for both the PGDS and the MPS. The client request API was then used to:

- invoke the FDIR system to diagnose a problem specific to the PGDS.
- invoke the functionally redundant FDIR systems to diagnose a problem in the MPS. Different client request profiles were specified to invoke different numbers and kinds of FDIR servers for the MPS.
- invoke the system-level FDIR service to isolate a problem that could originate in either the MPS or PGDS.

This last test demonstrates a global management capability, in which reports of anomalies trigger coordinated system-wide FDIR activity. (Appendix A lists a partial execution trace for this particular Server Group control test.) Server Group architectures are attractive for complex applications such as distributed operations support because of their modularity and extensibility. New capabilities, such as FDIR applications for different MPSs, can be integrated incrementally, without having to modify (the knowledge bases of) existing group members. All knowledge concerning subsystems and their interrelationships can be isolated within the Server Group, via service decomposition and result combination behaviors. These behaviors, if they are sufficiently complex, may themselves be implemented as system-level intelligent applications.

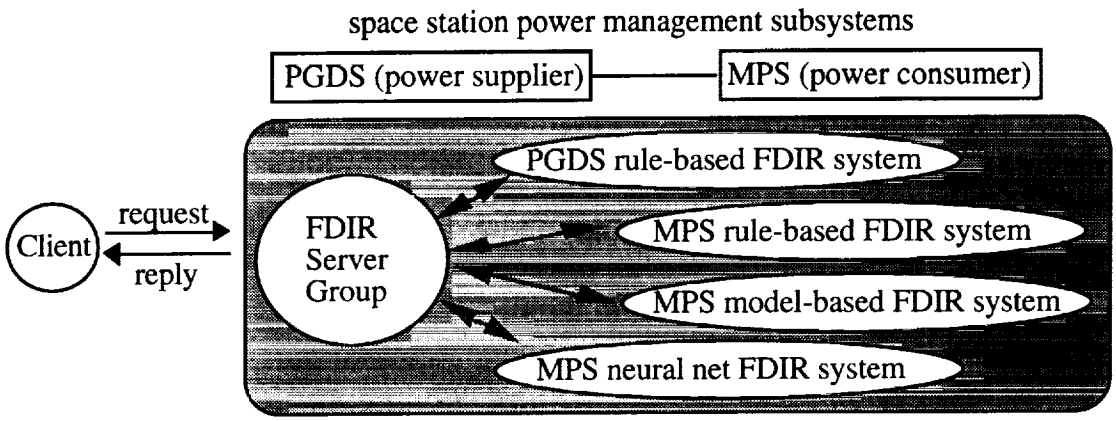

Figure 9. Demonstration scenario for the NDSL Server Group

CONCLUSIONS

The NetWorks! Server Group provides a generic, object-oriented engine for coordinating distributed one-to-many client-server interactions. Major application areas include distributed decision and operations support, data analysis, concurrent engineering, process control, and office automation. The Server Group reuses and specializes a one-to-one client-server request broker model. The extensions adapt process group transparency concepts to exploit functionally redundant servers and to manage client requests for complex services. High level APIs are used to specify server attributes, client request profiles, and request management behaviors specific to particular services. The Reliable Server Group extends the Server Group's model for managing replicated servers by incorporating a two phase commit protocol for reliability. Both models rely on a modular, communication middleware substrate for transparent application connectivity.

The NDSL Server Groups and related NetWorks! tools insulate developers and end-users from the distribution and heterogeneity of networked applications and their computing platforms. They also minimize the complexities associated with designing, implementing, maintaining, and extending the apparatus for coordinating one-to-many interactions among networked clients and servers.

ACKNOWLEDGMENTS

The NetWorks! Server Group technology described in this paper has been developed with funding from the NASA Small Business Innovative Research Program under NASA contracts NAS8-39343 and NAS8-39905.

REFERENCES

R. M. Adler. (1992a) "Coordinating Complex Problem-Solving Among Distributed Intelligent Agents." *Telematics and Informatics*. Vol. 9. Nos. 3&4. pp. 191-204.

R. M. Adler. (1992b) "Object-Oriented Tools for Distributed Computing." *NASA Proceedings for Technology 2002 Conference*. NASA CP-3189. pp. 146-155.

K. Birman. (1993) "The Process Group Approach to Reliable Distributed Computing." *Communications of the ACM*. Vol. 36. No. 12. pp. 36-53.

A.H. Bond and L. Gasser. eds. *Readings in Distributed Artificial Intelligence*. Morgan-Kaufmann. San Mateo, California. 1988.

S. Ceri and G. Pelagatti. (1984) *Distributed Databases, Principles and Systems.* McGraw-Hill. New York.

J. R. Corbin. (1991) *The Art of Distributed Applications: Programming Techniques for Remote Procedure Calls.* Springer-Verlag. New York.

G. Coulouris and J. Dollimore. (1988) *Distributed Systems: Concepts and Design.* Addison-Wesley, Reading, Massachusetts.

L. Gasser and M. Huhns. eds. (1989) *Distributed Artificial Intelligence, volume II.* Morgan-Kaufmann. San Mateo, California.

M. F. Kasshoek, A. S. Tanenbaum, and K Versteop. (1993) "Group communication in Amoeba and its applications." *Distributed System Engineering.* Vol. 1. pp. 48-58.

L. Liang, S. Chanson, and G. Neufeld. (1990) "Process Groups and Group Communi-cations: Classifications and Requirements." *IEEE Computer.* Vol. 23. No. 2. pp. 56-66.

Object Management Group and X/Open. (1991) "The Common Object Request Broker: Architecture and Specification." OMG Document No. 91.12.1 (revision 1.1), OMG, Framingham, Mass.

M. Ringer, T. Quinn, and A. Merolla. "Autonomous Power System Intelligent Diagnosis and Control." (1991) *Proceedings of the 1991 Goddard Conference on Space Applications of Artificial Intelligence.* NASA CP-3110. pp. 153-168.

R. Strom and S. Yemini. (1985) "Optimistc recovery in distributed systems." *ACM Transactions on Computing Systems.* Vol. 3. No. 3. Aug 1985. pp. 204-226.

B. Walls, D. Hall, and L. Lollar. (1990) "Augmentation of the Space Station Module Power Management and Distribution Breadboard." *Proceedings 4th Workshop on Space Operations Applications and Research (SOAR '90).* NASA CP-3103. pp. 355-361.

APPENDIX A

--------- Partial Listing of FDIR SERVER GROUP Directory--------

service	systemlevel-powersys-fdir	labmodule-powersysfdir
server-agent	fdir-grp	rulelm
server-sys	markov	hertz
server-profile	completeness 4	completeness 3
	certainty 4	certainty 3
	speed 2	speed 3
decomp-algorithm	sys-fdir-decomp	nil
term-condition	nil (default)	nil
result-comb-alg	combine-sys-fdir-results	combine-mps-fdir-results

GROUP-MEMBERS
(nntwklm hertz) (mbrlm markov) (rulelm hertz) (pgds markov)

--------- TRACE OF CLIENT REQUEST TO DIAGNOSE SYSTEM-LEVEL PROBLEM --------
Client Requesting System-Level Diagnosis for Power Mgmt. System...

FDIR-GRP Filtering Group Candidates for Service Type systemlevel-powersys-fdir...
Client Request Profile:
Max Servers: 10000 Discrim. Threshhold 10000 *;;;; defaults - use all candidate servers*
Completeness 0 Precision 0 Certainty 0 Timeliness 0 *;;; ignore all server attributes*
Candidates: ((systemlevel-powersys-fdir fdir-grp))
Applying Decomposition Algorithm.... *;;; Invoke Decomposition*
Substituting services (primary-powersys-fdir labmodule-powersys-fdir) *;;; Subtask servers*
Dispatching service systemlevel-powersys-fdir for SGROUP execution...
Symptoms: ("PDCU-B Bus-A LC1 RPC 3" "PDCU-B Switch RBI.3/1 Power 2.38")
Routing primary-powersys-fdir task to Agent APGDS... *;;;Send to subserver 1*
Routing labmodule-powersys-fdir task to Agent ARULELM... *;;;Send to subserver 2*
...
Agent APGDS Responding to Request for Service primary-powersys-fdir... *;;; PGDS Server*
injecting symtom data into PGDS ... diagnosing... diagnosis completed.
APGDS extracting diagnostic conclusions from PGDS:
Leakage-path high-to-low transmission-line RPC.3/6 load
Posting result to FDIR-GRP...
...
Agent ARULELM Responding to Request for Service labmodule-powersys-fdir... *;;; MPS Server*
injecting symtom data into RULELM ... diagnosing... diagnosis completed.
ARULELM extracting diagnostic conclusions from RULELM:
Low-impedance-short cable below switch
Low-impedance-short switch output of switch
Low-impedance-short switch input of a lower switch
Posting result to FDIR-GRP...
...
FDIR-GRP Responses collected for systemlevel-powersys-fdir request
Applying Result Combination Algorithm COMBINE-SYS-FDIR-RESULTS...
Conclusions: *;;;; result combination algorithm - logical union operation*
PGDS "Leakage-path high-to-low transmission-line RPC.3/6 load"
RULELM "Low-impedance-short cable below switch"
"Low-impedance-short switch output of switch"
"Low-impedance-short switch input of a lower switch"