Process Groups and Group Communications:

Classifications and Requirements

Luping Liang, Samuel T. Chanson, and Gerald W. Neufeld
University of British Columbia

Group communication is an operating-system-level abstraction that offers convenience and clarity to the programmer. Currently, only a few operating systems support this abstraction. However, a lack of understanding of group communication requirements with respect to different classes of applications may be equally responsible for the abstraction's not being widely used.

This article examines different distributed applications and outlines their requirements for group communication support. On the basis of internal structures and external behavior, we classify groups into different categories and discuss their properties.

To design a general, coherent, and integrated group communication system, we must understand basic application requirements. This classification of group applications can be an important tool.

In distributed computer networks, multicast (one-to-many communication) is a message transmission mechanism that delivers a message from a single source to a set of destinations. Special cases of multicast are unicast (one-to-one communication) and broadcast (one-to-all communication). As Figure 1 shows, a single multicast transmission delivers a message to a set of destinations in parallel, allowing the receivers to process the message concurrently.

In contrast to multicast, an abstraction of network communication, group communication is an operating-system-level abstraction. A group is a composite of objects sharing common application semantics, as well as the same group identifier and/or multicast address. (Multicast exists at the medium-access sublayer with multicast hardware. The mapping between group identifiers and multicast addresses is implementation dependent and not necessarily one-to-one.) Each group is viewed as a single logical entity, without exposing its internal structure and interactions to users.

Generally, objects are grouped for (1) abstracting the common characteristics of group members and the services they provide, (2) encapsulating the internal state and hiding interactions among group members from the clients so as to provide a uniform interface to the external world, and (3) using groups as building blocks to construct larger system objects. Passing messages to a group is generally called intergroup communication.

Group communication offers improved efficiency and convenience because

(1) it delivers a single message to multiple receivers by taking advantage of a network's multicast capability, thereby reducing sender and network overhead;

(2) it provides a high-level communication abstraction to simplify user programs in interacting with a group of receivers; and

(3) it hides from applications the internal coordinations of a group (for example, membership changes).

Group communication is best supported by network multicast. It can also be emu-

0018-9162/90/0200-0056$01.00 © 1990 IEEE

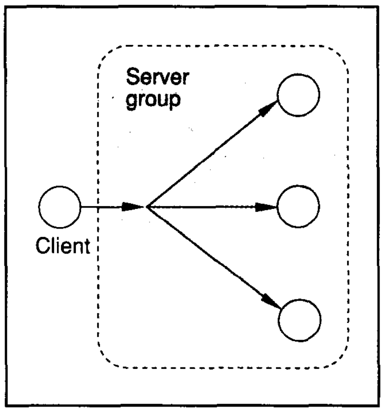


Figure 1. One-to-many communications.

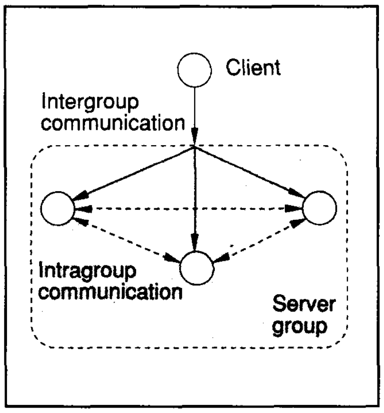


Figure 2. Inter- and intragroup communications.

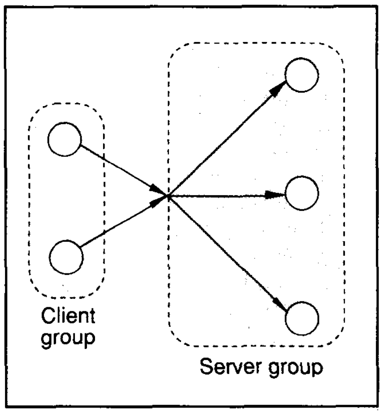


Figure 3. Many-to-many (group-to-group) communications.

lated by one-to-one interprocess communication or simulated by network broadcast. However, in the former case, not only is the first advantage lost, but a sender must also keep track of every group member; thus, groups are no longer self-contained. With network broadcast, extra overhead is required because all machines must examine every message regardless of its destination; furthermore, the communication is less secure, since group messages can be seen outside a group.

Although multicast was introduced a few years ago,[1] few applications take advantage of group communication. The reasons for this are (1) a lack of understanding of group communication requirements with respect to different classes of applications; (2) the fact that few systems provide sufficient group communication support at the operating system level to meet those requirements; and (3) until recently, a lack of multicast hardware support.

This article deals with the first part of the problem. We classify groups on the basis of their structure and behavior and present a uniform treatment of group transparency. Readers interested in the second part should refer to the literature.[1-7] Other work in the area includes that of Mockapetris,[8] who presents a general analysis of multicast mechanisms at the network level rather than at the application level. Hughes presents a multicast taxonomy based on the number of replies to each multicast request.[9] However, there is no general examination and classification of group communication requirements from the application's point of view.

We will define the process group model and classify it on the basis of the homogeneity of the structure among group members. Then we will examine different distributed applications and their requirements for group communication support, classifying groups according to their behavior. We will also discuss the relationship between the two classifications.

Process group model

The concept of a process group is not new. V[1] and Isis[2] defined a process group as a set of processes grouped to cooperatively provide a service. This definition, although broad, does not adequately characterize why or how the processes are grouped and hence does not provide sufficient information for classification.

We refine the process group definition as follows: First, we define an object as a set of variables and a set of operations on those variables. We define an object group as a set of objects sharing one or more common characteristics (internal state), interacting and coordinating among themselves to provide a uniform external interface. Second, because each object is generally maintained by a manager process and can be accessed only through a request to its manager, we define a process group corresponding to a given object group G as the set of manager processes maintaining the objects in G.

Process group members control the way resources in the object group are accessed and may have to coordinate among themselves to maintain state consistency in the object group. Members also interface with users of the resources they maintain. Group messages are generally sent to process groups.

This model provides a better understanding of group communication and allows classification on the basis of process group structure. For simplicity, we will use the term "group" to refer only to process groups.

In distributed systems, machine boundaries prevent processes on different hosts from physically sharing memory. In the following discussions, unless stated otherwise, we assume that interaction with object groups occurs through intergroup communication at the process level. The underlying network can be either a local area network or an internet. We make no assumption regarding the implementation of group communication.

A group is "closed" when only its members are allowed to send requests to it; otherwise, it is "open." In this article we will use the open model, as it is more general and corresponds to the client-server model commonly used in distributed operating systems.

In the client-server context, the process invoking an operation is called a client and the process receiving and processing the invocation is called a server. A process can play both client and server roles, depending on its communication context. This client-server model can be extended into group communication; that is, client group and server group can be defined similarly. As Figure 2 indicates, communications between external clients and a server group are called intergroup communications, while internal communications among group members are known as intragroup communications. Intergroup communications could also occur between groups in the form of many-to-many communications, as Figure 3 shows.

Sending messages from a single process to a group is called one-to-many communication, and from a group to a single process is called many-to-one communication. Usually, many-to-many communications can be decomposed into one-to-many and many-to-one communications.[4] Also, as Figure 4 shows, a process can maintain multiple objects that may belong to different object groups. Therefore, a process can belong to multiple groups, and process groups can overlap.

Structural classification

Viewed as a collection of object managers, a group can be classified on the basis of the homogeneity of the internal state of the objects maintained and operations supported by each group member. An object manager can be characterized by

- *Application-level objects* — the set of objects maintained by the process. Their value determines the application-level internal state of this process.
- *Application-level operations* — the set of operations that can be executed on the above objects. Other processes can modify the value of these objects only by invoking these operations through this manager.

Operations on the objects are what define the services a process provides. Because services of a process are accessed through interprocess communication in message-passing systems, operation executions and process state transitions are stimulated by the events of message arrival.

A *selection rule* of a group G is a set of criteria for selecting objects forming G and is determined by G's application. For simplicity, we assume that an object is in the object group G if and only if the object satisfies G's selection rule, and that a process p is in the manager group of G if and only if p maintains at least one object satisfying that rule. The services a group provides are implemented by the group members cooperatively and are accessed by clients through intergroup communication. A group G can be characterized by

- *Group objects* — the subset of objects maintained by each member process in G, which satisfies G's selection rule.
- *Group operations* — the set of operations on the above objects.

Depending on how each group member

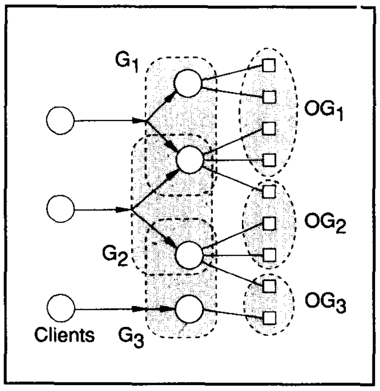


Figure 4. Communication pattern for overlapping groups; G_i = process groups, OG_i = object groups.

implements and maintains objects and operations, a group can be placed in one of four categories:

(1) *Data and operation homogeneous* (DOH): Every member in a DOH group maintains a complete replica of the set of group objects and implements an identical set of operations on these objects. To guarantee consistent external behavior, a DOH group maintains consistency among replicas of the objects and requires every member to execute exactly the same sequence of operations. DOH groups are used mainly to increase service reliability and availability. Examples are groups in Isis[2] and troupes in Circus.[4]

(2) *Operation homogeneous only* (OHO): In an OHO group, the object space is partitioned among group members, with each member maintaining only part of the global group state. Object space partitions may overlap. Also, every member supports an identical set of operations on its portion of the objects. However, when an operation is invoked on an object, only members with the relevant object need to perform the operation. OHO groups are used mainly to distribute the work load among group members. Each member can maintain the integrity of its own objects independently of other members. The distributed name service, discussed later, is an example of the OHO group.[10]

(3) *Data homogeneous only* (DHO): Members in a DHO group share a set of objects by sharing the same address space on a single machine, or in some other distributed manner (for example, data replication). Each member supports a set of operations on the same objects. These operations may or may not be identical to those of other members. Upon invocation of a group operation, members may act differently. For example, a coordinating member accepts an operation invocation from a client and accomplishes the task through internal cooperation with other group members. The role of coordinator need not always be played by the same member. Also, members in a DHO group must synchronize themselves to serialize concurrent updates to the objects. This requires an underlying mechanism similar to that for DOH group support. The usual purposes of a DHO group are to provide group services cooperatively via a set of worker processes and to simplify the design, implementation, and interface of the service by masking member cooperation from external observation. Examples include the team in V[11] and the primary-secondary replication scheme.[6,12]

(4) *Heterogeneous* (Het): As far as the group application is concerned, the objects and operations each member implements and maintains could both be heterogeneous. There may or may not be cooperation among group members, and their internal states may be completely independent of one another. Rather than encapsulate interactions among members to provide a cooperative group service, heterogeneous groups facilitate system control and simplify interactions between the client and server groups. Electronic mail distribution lists, computer conferencing, news groups, and distributed process control are applications of heterogeneous groups.

Behavior classification and requirements

According to their external behavior, distributed process groups can be classified into two major categories: deterministic and nondeterministic. The former groups are used mainly in replicating data and services to enhance reliability, while the latter are used mainly in distributing data and work load among multiple servers to improve information availability and resource sharing. More complete definitions of the two categories appear later.

Basically, a deterministic group requires high reliability in group communications to maintain strong consistency among members. Such groups are "heavyweight" in the sense that they require complete group membership information and atomic, consistently ordered group interactions. In contrast, nondeterministic groups are "lightweight," since they need only basic datagram multidelivery transport support. Inconsistencies and unreliable group interactions are handled in an application-specific manner, resulting in more flexible and efficient — but more complex — application programming. Whether a group is deterministic or nondeterministic depends solely on its application, not on its structural characteristics.

Distributed process groups can be classified into two major categories: deterministic and nondeterministic.

Deterministic groups. A group is deterministic if each member must receive and act on a request. (The term deterministic is used here to characterize the relationship among group members and is less restrictive than when used by some authors to mean only one possible execution of a single procedure.) This requires coordination and synchronization among group members. In most deterministic groups, member processes are *equivalent*;[4] upon receiving the same request in the same state, the same procedure is invoked, and every member transfers to the same new state and produces the same response and external effects.

Let's look at some deterministic-group applications in terms of their basic characteristics and communication requirements.

Replicated file systems. In a fully replicated file system, all file servers constitute a group. Files are replicated at every file server to enhance file availability and reliability. The two common methods supporting replicated file systems are *peer-member* and *primary-secondary*.[3] In the peer-member scheme, all members in a file server group are identical, and group communication and coordination occur between the client and all members. In the primary-secondary scheme, a primary member handles the communication interface between clients and the group, and group communication and coordination occur between the primary and all secondaries inside the group. A fully replicated file server group is a deterministic DOH group if the peer-member scheme is used; it is a deterministic DHO group if the primary-secondary scheme is used.

Replication transparency is an important characteristic of these systems. File system clients usually prefer a single file image regardless of whether a file is implemented by one or many servers. The file abstraction as seen by a client is called a logical file image, and operations on it are called logical operations. The physical file copies maintained by the file servers are known as file replicas, and operations on them are called physical operations. Reliability requires that file replicas be kept consistent at all servers, so that files are always available to clients as long as at least one file server is functioning. Availability requires that users be able to read the files with minimum latency — for example, to get a consistent copy of a file from the closest functioning server.

Both reliability and availability imply that file replicas must be consistent. Thus, every logical file update must be atomic, that is, either executed by all servers or by none, and a client must be informed of whether or not its update is completed. Furthermore, because different sequences of the same set of update operations can result in different file states, all servers must execute exactly the same sequence of operations with respect to each logical file. Two logical operations, op_1 and op_2, are in conflict if they are data dependent; since they manipulate overlapping logical data, the execution order affects the results. Two conflicting operations collide if executed concurrently.

Consider two clients, C_1 and C_2, who issue a LockFile request independently to acquire a lock on the logical file F. If the two requests are not consistently ordered at all server group members, some servers may allocate the physical locks on their physical replicas of F to C_1, others to C_2, depending on whose request is executed first; thus, a deadlock may occur. Clearly, collided operations must be ordered; the order can be arbitrary as long as it is consistent at all member sites. This ordering of collided operations is called absolute ordering.[2]

File servers can be added or deleted dynamically. Clients expect the file server group to coordinate internally to hide membership changes from external observation. This type of internal coordination includes keeping consistent name binding between a group identifier and a set of server process identifiers, and bringing a new server state up to date. Satisfying this requirement not only simplifies the group interface to clients but also increases flexibility in file system configuration. Since a host cannot always distinguish between network partition and host failure, file-system-level consistency requires that both clients and servers be notified when either failure occurs, and that some higher level consistency-control algorithms be used to handle the failure.

Replicated program executions. Replicated program execution is another example of the deterministic DOH group. In some distributed applications, a major concern is the resiliency of computations. Programs can be decomposed into abstract data type modules, each having a set of internal states and a set of procedures manipulating the states. Modules are replicated on multiple sites,[4] and replicas of a module form a group. Group members may represent different implementations of the same abstract data type written by different programmers (subject to the constraint of equivalent deterministic behavior). Thus, module reliability can be enhanced by both replication and multiversion programming. Multiversion programming tends to reduce program design faults (assuming fail-stop behavior), while replicated module execution tends to make the module runtime execution robust. A procedure call to a module can proceed as long as at least one group member is functioning. An application programmer would expect the syntax and semantics of a replicated procedure call to remain the same as those of a nonreplicated call, that is, replication transparent.

Consistency in replicated procedure calls requires that the group members be deterministic and equivalent and that every member execute the same sequence of calls. Application programmers are responsible for ensuring that modules are deterministic and equivalent. The group system, on the other hand, must guarantee atomicity and ordering (defined in the previous section) to each procedure call, as well as replication transparency. For each replicated procedure call from a client group to a server group, each client group member makes a one-to-many call to the server group, and each server group member accepts many-to-one identical procedure invocations (resulting from one call per client group member) and makes a one-to-many reply to the client group after execution. Each client group member then handles many-to-one replies from the server group. These multiple calls and receptions are best handled in the group communication layer so that client/server programmers need not be aware of multiple entities in the caller/callee group.

An advantage of making replicated procedure calls at the module level is that the degree of replication can be adjusted dynamically according to the functions of different modules, thus optimizing system performance and reliability. In other words, more copies could be made for critical modules to enhance their reliability, while fewer or no replications would have to be made for less important modules. Also, dynamically changing group membership allows system auto-reconfiguration to become transparent to applications as group members fail and recover at runtime. However, this dynamic group membership makes it difficult to bind a group name to a set of modules.

A replicated program execution often happens within a single local area network; therefore, network partition failure is unlikely. When any group member fails, the group is expected to reconfigure itself autonomously, making partial failures transparent to client groups.

Distributed industry process control. Distributed process control is an example of the deterministic Het group. Imagine a simple distributed industry process control environment in which the temperature in a reaction container is to be controlled. A sensor measures the current value of the temperature, and a control panel displays that value to human operators, records the history of the sensor signals, and allows an operator to set the control parameters. A set of controllers manages the flow of cooling fluid into the container by opening or closing valves. The sensor can view the console and the controllers as a group. When a control parameter diverges from the preset value, the sensor multicasts the measured result to the group so that the console informs the operators and the controllers open or close the valves to adjust the flow automatically.

This group is obviously heterogeneous because each member, though driven by the same sequence of signal stimuli, maintains completely different objects and performs different operations. These components are grouped simply for convenience of communication; they are identified by a single receiver ID in each group message. If the underlying network supports multicast, only one copy is transmitted for each sensor signal, saving network bandwidth and speeding the signal processing.

Although member states have no application-level consistency requirement, group communications have reliability requirements. Unless the sensor fails, its signals must be delivered reliably to all active members in the same order as generated. Also, each signal from the sensor must be delivered by a predefined deadline or the signal will become obsolete. The sensor expects no reply from the group. Any individual member failure must be detected quickly, and operators must be notified to repair the failed component. However, before recovery of the failed component, the remaining group members are expected to continue fulfilling their duties even though the complete group service may not be available.

E-mail distribution list and computer conferencing. Electronic mail distribution lists and computer conferencing are two other examples of the deterministic Het group. People registered in the same distribution list or conference constitute a group. Grouping is for convenience of communication; for each message, a sender prepares a single copy and performs one send operation. Generally, a sender does not know who participates in the distribution list, because registrations are handled by an independent authority. The sender simply sends the message to the list's address, which has the same syntactic format as other single-user addresses and which logically includes all participants. The same is true for computer conferencing, except that conferences are normally closed groups in which only participants can send messages to the conference. In contrast, a distribution list is open to anyone with access to the list address.

Assuming registration and network connections are set up properly, messages to a conference are expected to be delivered atomically. If any participant receives a group message, other members in the group should also receive that message within a bounded period. However, for each message delivered, the sender may receive replies from recipients, either in the form of private one-to-one correspondence or as follow-up discussions to all conference participants. In a follow-up message m, the speaker makes a point on the basis of all the messages related to m that he or she has received; these messages are called the context associated with m. For other participants to understand m properly, they must be able to reference its context.[7] A recipient should not see a message without having also received its context. This dependent relationship is sometimes called a causal relationship[2] and defines a partial ordering among messages submitted to a conference.

In both distribution lists and conferencing, absolute ordering is not required. Concurrent messages usually can be delivered in arbitrary order because they are not context related and because the participants' states are not message dependent. When necessary, the conference chair determines the order of concurrent messages. Sending and receiving messages can be concurrent and asynchronous for each participant. Because incoming messages may affect the content of outgoing ones, receiving may be given higher priority than transmission.

In conferencing, a follow-up message may become a *competitive orphan* (explained later) because of the asynchronous nature of the communication pattern. In this situation several people may respond identically to the same message before seeing each other's responses. Fast delivery makes this problem less likely but does not eliminate it entirely.

Membership changes should not affect group communications. A new member normally becomes up to date by reading the conference bulletin board or the discussion archive. The failure of a member is normally treated as departure from the group.

Distributed databases. A deterministic Het group can be used within a distributed database. A distributed database model has a transaction manager (TM) and a data manager (DM) on each site. Each TM accepts user requests and translates them into commands for DMs. Each DM maintains part of the database stored at its site and may concurrently execute transactions from multiple TMs. A transaction group consists of all DMs participating in the same transaction. This group is heterogeneous because each DM maintains a different part of the database and each may respond differently (accept or reject the precommit from the TM) according to its local status. Messages from the coordinating TM are delivered atomically to the transaction group.

For concurrency control and failure recovery, a two-phase commit protocol (2PC) is normally used at the end of each transaction. When the coordinating TM and all cohorts (DMs) that know the decision (commit or abort) fail, the standard 2PC will be blocked until some processes recover. A simple nonblocking 2PC can be designed on the basis of ordered atomic group communication.[3]

In the absence of a coordinating TM failure, the protocol proceeds as the normal 2PC. When the coordinating TM fails, a communication-layer-generated failure notification is broadcast to the group, and upon receiving this notice each DM aborts the transaction. This protocol requires not only an atomic but also an ordered group communication mechanism to guarantee (1) that every DM in a transaction group will receive exactly the same sequence of instructions from the coordinating TM, and (2) that if the coordinating TM fails after broadcasting commit, its failure notification is delivered either before or after its commit message consistently and atomically to all DMs, allowing them to either commit or abort the transaction consistently. Although the membership of a transaction group is governed by the coordinating TM, and a DM failure is detected by the TM during the execution of 2PC, the communication layer still needs this information to achieve atomicity. Inconsistent replies from DMs are handled by the coordinating TM itself rather than by the group mechanism.

Summary of requirements. The above analysis of deterministic group applications makes clear that group transparency is a desirable property allowing a client to treat a group's service as if it were provided by a single server. A single call is made and a single result, if any, is expected from the server group. Also, a server maintaining group object replicas need not know that another coserver exists, and thus application programmers need not be aware of group coordination.

Group transparency for deterministic groups must satisfy the following requirements:

(1) *Communication transparency.* Communication transparency consists of two aspects: atomicity and ordering.[2]

• *Atomic message delivery.* An atomic group message is either received and processed exactly once by all members in the recipient group, or by none at all. Atomicity hides a partial group communication failure by converting it into a total failure.

• *Application-level absolute ordering.*

In both distribution lists and conferencing, absolute ordering is not required.

The delivery order of messages to group members needs to be synchronized if the order will affect the result. Every member of a group must see the same sequence of requests on dependent data and adjust its internal state accordingly. Also, in the case of colliding requests, the common members of two overlapping groups must see a consistent "combined" sequence of requests to both groups.

(2) *Reply-handling transparency.* Because client and server interactions normally follow the request-response pattern, reliable multicast from client to server is not enough. Replies from a group also must be collected and processed properly to achieve group transparency.[2,4] For each group request, there exists a potential for multiple responses from a server group. These responses may or may not be identical. Reply-handling transparency guarantees that a client need not be aware of the multiple replies to its request. It sees a single reply without having to be concerned about how this reply is derived from others (for example, by weighted voting).

(3) *Naming transparency.* This involves dynamically and transparently binding group members to a single name. Group naming consists of two parts: mapping a logical service name into a group of servers, and allowing group membership to change dynamically. A group view is a snapshot of the group membership at a particular instant in time. It is maintained by each of the concerned parties, be they group members, system name servers, or any client needing to make decisions on the basis of the group view. A group view changes as members fail and recover, or are inserted and deleted, in parallel with other group message activities.

Since atomic group interactions rely on a consistent group view to verify that all active members have confirmed reception of each atomic message, group view changes must be detected in a consistent manner. It is convenient to serialize group view changes consistently with respect to other group message activities. Isis group members achieve this by having a system-generated announcement, issued on behalf of the failed member, follow all its prefailure messages. This announcement arrives at every member in the same order with respect to the other group messages, so that members all see the same sequence of group view transitions at virtually the same time. Therefore, it is guaranteed that no message will arrive from a failed member once its failure has been announced.[2] If a member recovers, its state must be brought up to date with a consistent snapshot of the group's internal state, and all concerned parties should perceive the new member's existence before receiving a message from it.

(4) *Failure transparency.* Depending on a group's purpose, either clients and server group members are notified of the failure to take application-level recovery actions, or a member failure is hidden from the clients. In the latter case, the failure may be presented as a complete group failure, or other group members may take over the role of the failed member. The technique chosen depends on the group's function. When a deterministic group is used to enhance data reliability, as in replicated file systems or databases, strong consistency is required among the group object replicas. Therefore, the group consistency control strategy may exaggerate a partial failure as total; if any member fails or the network is partitioned during a group operation execution, the operation cannot succeed until the failure is recovered or the group is properly reconfigured. However, when a group is used to enhance service reliability, as in a replicated procedure call, maintaining service to clients is important. During recovery of the failed member, remaining active group members should continue fulfilling their duties to keep damage to a minimum.

(5) *Real-time requirement.* We can measure multicast message delay in terms of distribution time — the time taken for all operational members in a group to receive a multicast — or in terms of completion time — the time taken for the sender to learn that all destinations have received its message reliably.[8] In deterministic group applications, consistency is more important than efficiency. When trade-offs between the two are required, system support often gives priority to consistency. In real-time systems, on the other hand, messages

usually must be delivered within a client-specified deadline; otherwise, a message is deemed obsolete and a timing fault is triggered.

In one-to-one interprocess communications, a server can simply ignore a timing-fault message or respond to the client with an operation failure. In multicast, some servers may receive the message on time while others see a timing fault, even though atomicity and ordering are guaranteed. When this occurs, servers in the recipient group must coordinate to act consistently on each timing-fault message to guarantee consistency. Multicast distribution time should be bounded so that group actions can be scheduled to occur atomically and simultaneously in virtual time at all group members.[13]

Nondeterministic groups. Deterministic groups require strong data and behavior consistency, and synchronization among all members. Nondeterministic groups assume that their applications do not need such strong built-in consistency, and they relax it in various application-specific manners. Nondeterministic group members generally are not equivalent.[4] Each member may respond differently to a group request, or not respond at all, depending on the individual member's state and function. Normally, either member states of a nondeterministic group are unaffected by processing user requests, or they are not necessarily consistent. Either requests to such a group do not require all group members to act, or missing requests can be detected and recovery completed within the application.

Because of the relaxed consistency, maintenance overhead for the group is generally lower than for a deterministic group. Whether a group should be implemented as deterministic or not depends on the application requirements.

Next we will describe some nondeterministic group applications in terms of their basic characteristics and communication requirements.

Distributed clock service. The time-of-day service in the V system exemplifies a nondeterministic DOH group,[11] in which all group members implement the same set of functions and play the same role in the service. Although all members are supposed to maintain identical objects, the state of these objects may differ when a partial group communication failure results in some members not receiving a group request.

Failure of any name server does not stop activity at other servers.

In the distributed time-of-day service, every station periodically receives a clock tick from a central clock. Between clock ticks, each machine's clock replica simply caches the latest time stamp from the central clock and extrapolates forward using its local clock. Clock drifting can be corrected by the next clock tick. Time requests are handled using the locally extrapolated time values. Should any clock update message be missed, the next clock update corrects it. Although the time value stored in the local clock may not be absolutely correct, it is accurate enough for most non-time-critical applications.

Similarly, many applications that replicate data do not require absolute consistency. The required level of consistency is obtained by using application semantic knowledge and assuming that the client can detect, recover from, or tolerate inconsistencies. This reduces communication complexity and promotes efficiency. The nondeterminism in these applications stems from the fact that group members may maintain an inconsistent or inaccurate global group state. Applications of this type need only an efficient and "best effort" multicast mechanism.

Distributed name service. In a distributed name service, a set of name servers operate at several machines in the network. In some designs the global name space is partitioned, and a different name server maintains each partition.[10] This type of distributed name server forms a nondeterministic OHO group, because every member performs the same set of operations. As an example, each object in the operating system Clouds[10] is assigned a logical system name. A mapping function w maps a system name onto a multicast address $A = w(S)$. Each station maintains a multicast address table for the objects stored at the node. To locate an object O, a user provides its system name S. The client host first calculates O's multicast address $A = w(S)$ and uses A as an index to search its local object directory. If the object is not found, a multicast remote procedure call is invoked on the address A. Nodes with A in their multicast address table accept the remote procedure call, search the local object directory for the object with the system name S, and reply if they find it. In this way, all nodes share the overhead of maintaining, locating, and migrating objects.

Group transparency is an important property expected by both clients and servers. A client locates an object by submitting a name look-up request to the name service. It is irrelevant to the client which server responds or whether single or multiple servers handle the request. Each server manages its assigned portion of the name space and responds only to requests for objects it knows. It need not be aware that other servers exist.

Nondeterminism in distributed object naming results from the global group state's being partitioned among members; therefore, a client does not know which server will perform its request. A LookUp need not be multicast to name servers atomically, because it is an idempotent operation and does not alter the name servers' state. Note, however, that the name-binding update in the name servers is a deterministic operation, and all replicated servers, if any, must execute the update operations atomically and in the same order in which names are bound.

Grapevine[12] adopted another way of supporting a distributed name service. It exemplifies the nondeterministic DHO group in which the name space is fully replicated at each name server. However, a server may play primary or secondary roles in name updates, even though every server supports the same set of operations. The Grapevine update algorithm does not guarantee a consistent view at all name replicas, because name update broadcasts are neither atomic nor ordered. A client deals only with its local name server without seeing the whole name server group. Clients can detect and correct inconsistent and stale names in an application-specific manner.

Both examples should make clear that failure of any name server does not stop activity at other servers.

Contract bidding. Contract bidding, another example of the nondeterministic OHO group, is a technique for resource sharing and load balancing among stations in a server pool. Clients submit job requests for servers to complete. The specific server station and the order of execution do not matter. Server group members do not maintain a global group state. All worker stations are functionally identical and respond to idempotent service contract bids only on the basis of their local state. Upon completing a task, servers return to the ready state for the next job assignment.

The number of available server stations is always changing, as is their current load. To optimize overall system throughput and mean response time, tasks should be scheduled to keep all servers equally busy. Scheduling is usually done in one of two ways: In a client-initiated scheme, a client posts its task requirement on the network for server stations in the pool to bid on. An available server responds with its current load condition, and the client chooses the proper server to complete the task. In a server-initiated scheme, potential clients form a group, and an available server posts its request for loading to the client group. Each client responds with its task requests, and the server chooses the appropriate one to execute.

In both schemes a *competitive orphan* may be generated. In contrast to a *failure orphan*, generated when a server continues to execute a dead client's request, a competitive orphan is generated when server group members are not properly coordinated. (Failure orphans are a general problem in the client-server model, while competitive orphans are a special problem in the group communications context. We can view a failure orphan as resulting from a lack of coordination between the client and the server group; we can view a competitive orphan as resulting from a lack of coordination among group members.)

In the server-initiated scheme, a competitive-orphan request could be generated from a client that does not know whether its job request has already been carried out. Also, in the client-initiated scheme, multicasting a job request could trigger multiple concurrent executions at different servers, even when only a single execution is needed. The effect of competitive orphans must be cancelled or reduced to a minimum, and the competition losers must be notified so that they can participate in subsequent contract-bidding activities. Again, communication overhead between clients and server group members should be minimized; multicast should be efficient but need not be perfectly reliable. The failure of any server should not terminate the whole system. However, the client whose job was being executed by the failed server must be notified to resubmit its request.

The dominant communication pattern in nondeterministic group applications is request-response.

News propagation. News propagation in Usenet exemplifies the nondeterministic Het group. People subscribing to the same news group constitute a group, and the state and operations each subscriber performs differ. A news poster never knows who the members in a group are. News propagations need be neither atomic nor ordered. Respondents to a news article can use one-to-one personal correspondence or follow-up articles to the same or a different group. Each person then decides what to do with each news article and the follow-up comments. A subscriber can come and go at will; no synchronization is necessary.

Summary of requirements. Nondeterministic groups are intended to improve service performance. Through group transparency, a group service strives for the same simple syntax and semantics found in one-to-one interprocess communication. However, depending on the intended applications and the nature of the particular group, this transparency may be relaxed. Nondeterministic groups induce less overhead and generally have the following characteristics and requirements:

(1) *Communication transparency.* The dominant communication pattern in nondeterministic group applications is request-response. Usually, applications do not require absolutely reliable message delivery or message ordering. Interactions with nondeterministic groups are inherently asynchronous because (1) it is neither necessary nor realistic to expect a client to wait until all server group members are synchronized and ready to receive a request; (2) server group membership normally is not known to clients; and (3) a server may not receive the request at all. Data consistency is not a problem for various reasons: The application requests may be idempotent, group consistency may not be critical to the application, or the application may be able to detect and recover from inconsistency easily. Application programmers are given the flexibility — with the attendant complexity — of handling partial message failures.

(2) *Reply-handling transparency.* Multiple replies from a nondeterministic group may not be consistent. They may have to be handled by the clients in an application-specific manner rather than by the server group, as in deterministic groups; this sacrifices a certain degree of reply-handling transparency. Also, an application must provide its own time-out value for each multicast request because the communication layer itself does not know how long to wait for server group responses. When the timer expires, the application can decide whether to re-multicast or perform alternative actions. In deterministic groups, these actions are normally performed at the communication layer.

(3) *Naming transparency.* As with deterministic groups, applications prefer to use a logical name to address a group service rather than having to know each individual member. To achieve naming transparency, both clients and servers prefer to handle requests and replies independently of the number of servers. However, nondeterministic group membership can change dynamically and usually is not known to the active group members or to the communication layer.

(4) *Failure transparency.* Nondeterministic group member failure has different semantics from that of deterministic groups. When a member fails, other active members may transparently take over the uncompleted task to enhance availability, share service load, and reduce global communication traffic.

(5) *Competitive-orphan problem.* In a nondeterministic group a competitive orphan can be generated because of a lack of internal coordination among group members. Except in computer conferencing, most deterministic groups do not experience this problem, because a group request must be handled by every member synchronously while the client waits for a reply from every server.

Table 1 shows the differences between deterministic and nondeterministic categories.

Table 1. Differences between deterministic and nondeterministic categories.

	Deterministic	Nondeterministic
Communication	Normally requires atomicity and absolute ordering. Some applications require causal ordering.	Not necessarily reliable, nor ordered. Applications handle inconsistency.
Naming	Complete group view at communication layer is necessary. Membership changes must be synchronized with all other group messages.	Group view usually is not known to anyone, even to the communication layer.
Reply handling	If required, expect all members to reply. Group members handle inconsistent replies without the client's being involved.	Clients must handle inconsistent replies explicitly and applications have to provide the time-out parameter.
Failure handling	To enhance reliability, a partial failure is turned into a total failure in most cases.	To enhance availability, partial failures are usually hidden by active group members.
Others	In real-time systems, a timing fault may cause inconsistency.	A competitive orphan may arise due to a lack of proper coordination among group members.

Table 2. Sample applications based on the classifications.

	Deterministic	Nondeterministic
Data and operation homogeneous (DOH)	Fully replicated file systems (peer-member scheme) and replicated procedure call.	Applications such as the distributed time-of-day service in the V system.
Operation homogeneous only (OHO)	Partially replicated file systems.	Clouds' distributed name-server group and contract bidding.
Data homogeneous only (DHO)	Fully replicated file systems (primary and secondary scheme).	Grapevine's name-server group.
Heterogeneous (Het)	Distributed process control, computer conferencing, and E-mail distribution list.	News propagation in a news group of Usenet.

Discussion

Several existing systems support group communications. The Isis[2] and Circus[4] systems are intended primarily for deterministic replicated data objects or procedure module groups. Birman and Joseph[2] provide detailed design and analysis on protocols for atomic and ordered group communications. The V system[1,11] and several other experimental systems[5] support nondeterministic groups.

The two classifications of groups previously discussed are based on different criteria, one on structure, the other on behavior. By orthogonally projecting one over the other, as Table 2 shows, we hope to better understand how various group applications fit the classifications.

First, we can see that external uncertainties in most nondeterministic groups stem from the following facts: (1) objects are distributed only to a subset of group members, and the size and membership of this subset may not be known in advance; and (2) even when objects are fully distributed to all group members, applications do not require their values to be always consistent or accurate, and missing group messages can be tolerated.

Second, Table 2 shows that most deterministic group applications require messages to be sent atomically and in order, regardless of the homogeneity of group structure. Also notice that communication support for a deterministic DHO group is the same as for a deterministic DOH group. No matter how differently each individual DHO group member functions at the high level, to guarantee consistency among replicas, changes to the objects must be propagated atomically and in order.

Let's consider partially replicated file systems as an example of the deterministic OHO group in Table 2. File system reliability requires that for each update request, only those servers having the target file take action and respond. It is difficult to map a logical file onto an unknown number of file servers maintaining the physical replicas of the file. Thus there exists a level of inherent nondeterminism in the group communication. If we had a separate server group for each replicated logical file object, we would end up with a fully replicated deterministic DHO server group for each file. However, this may not be necessary, and having many dynamically changing groups could be expensive.

It would be preferable to install software filters at the client and the servers to eliminate this structural nondeterminism. Each server filter would discard requests for nonlocal files. The client filter would use some mechanism (perhaps by consulting a name server) to determine the membership of the implied subgroup — those having a copy of the target file — and to guarantee atomic message transaction with only this subgroup. Once the implied subgroup membership was determined, the group transaction could proceed atomically.

An alternative would be to have every file server reply to every request; those not knowing the target file would simply reply with a "null" message. The client would work on non-null replies using knowledge of the whole file server group membership to eliminate the above nondeterminism. This scheme trades extra host loading cost for structural determinism to gain reliability. It differs from broadcast in that (1) only file servers pay host-loading cost for each file access, and (2) as a system program, file servers are generally more trustworthy, and therefore file transactions can be made more secure.

Before designing a general, coherent, and integrated group communication system, we must understand how it will be used, that is, the basic application requirements. We analyzed different types of groups, along with their potential applications, and classified group applications into two major categories: deterministic and nondeterministic. Orthogonally, according to the structure, process groups can also be classified as data operation homogeneous, operation homogeneous only, data homogeneous only, or heterogeneous.

A basic conclusion from this analysis is that group transparency is important and desirable. When integrated into the underlying group support, it simplifies the interface between server groups and their clients by hiding from the clients, as much as possible, the membership of server groups and the interactions among group members. This enables designers of clients and servers to concentrate on the problems to be solved — as they do in the unicast environment — without concern for coordinating multiple servers. Group transparency is manifested in group communication, group naming, multiple-reply handling, group view change, and partial failure.

We hope this classification framework and analysis will enhance the understanding of process groups, group communications, and some applications, thus aiding designers working with these mechanisms.■

Acknowledgments

We are grateful for suggestions from the anonymous referees, which led to a more concise presentation. This work was made possible by grants from the Natural Sciences and Engineering Research Council of Canada.

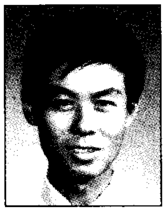
References

1. D.R. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel," *ACM Trans. Computer Systems*, Vol. 3, No. 2, May 1985, pp. 77-107.
2. K.P. Birman and T.A. Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. Computer Systems*, Vol. 5, No. 1, Feb. 1987, pp. 47-76.
3. J.M. Chang, "Simplifying Distributed Database Systems Design by Using a Broadcast Network," *Proc. ACM SIGMOD*, June 1984, pp. 223-233.
4. E. Cooper, "Replicated Distributed Programs," *Proc. ACM 10th Symp. Operating Systems*, Dec. 1985, pp. 63-78.
5. L. Hughes, "A Multicast Interface for Unix 4.3," *Software Practice and Experience*, Vol. 18, No. 1, Jan. 1988, pp. 15-27.
6. S. Navaratnam, S.T. Chanson, and G. Neufeld, "Reliable Group Communication in Distributed Systems," *Proc. Eighth Int'l Conf. Distributed Computing Systems*, June 1988, CS Press, Los Alamitos, Calif., Order No. 865, pp. 439-446.
7. L. Peterson, N.C. Buchholz, and R.D. Schlichting, "Preserving and Using Context Information in Interprocess Communication," *ACM Trans. Computer Systems*, Vol. 7, No. 3, Aug. 1989, pp. 217-246.
8. P.V. Mockapetris, "Analysis of Reliable Multicast Algorithms for Local Networks," *Proc. Eighth Data Comm. Symp.*, Oct. 1983, CS Press, Los Alamitos, Calif., Order No. 494, pp. 150-157.
9. L. Hughes, "A Multicast Response-Handling Taxonomy," *Computer Comm.*, Vol. 12, No. 1, Feb. 1989, pp. 39-46.
10. M. Ahamad et al., "Using Multicast Communication to Locate Resources in a LAN-Based Distributed System," *Proc. 13th Conf. Local Computer Networks*, Oct. 1988, CS Press, Los Alamitos, Calif., Order No. 891, pp. 193-202.

11. D.R. Cheriton, "The V Distributed System," *Comm. ACM*, Vol. 31, No. 3, Mar. 1988, pp. 314-333.

12. M.D. Schroeder, A.D. Birrell, and R.M. Needham, "Experience with Grapevine: The Growth of a Distributed System," *ACM Trans. Computer Systems*, Vol. 2, No. 1, Feb. 1984, pp. 3-23.

13. F. Cristian et al., "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," Tech. Report RJ4540(48668), IBM Almaden Research Center, Dec. 1986.



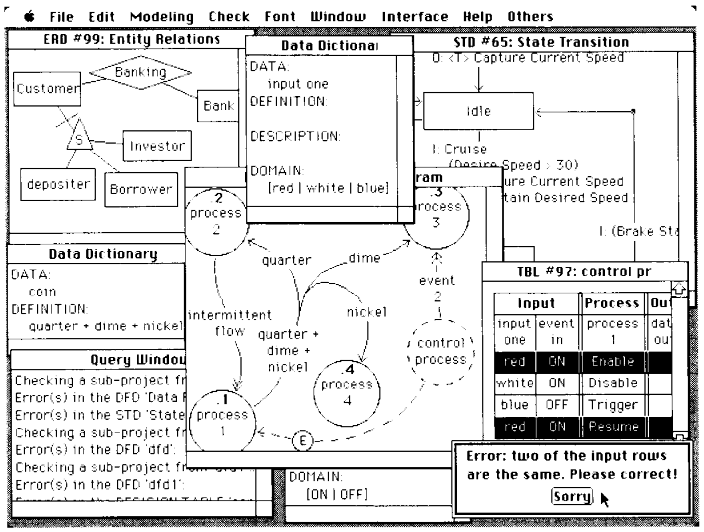
Luping Liang is a PhD candidate in computer science at the University of British Columbia. His research interests include distributed operating systems, computer communication networks, and performance modeling and analysis. He received an M.Math degree in computer science from the University of Waterloo in 1985 and a B.Eng. in computer engineering from Tsinghua University, Beijing, in 1982, where he also worked as a faculty member. He is a student member of ACM and the IEEE Computer Society.

Samuel T. Chanson is an associate professor in the Department of Computer Science at the University of British Columbia, where he is also a founding member of the Distributed Systems Research Group. He is researching distributed operating systems, computer communications, and performance analysis of distributed systems. Chanson organized and cochaired the first International Workshop on Protocol Test Systems, held in Vancouver in October 1988. He received a PhD in electrical engineering and computer sciences from the University of California at Berkeley in 1974. He is a member of the IEEE Computer Society.

Gerald W. Neufeld is an assistant professor in the Department of Computer Science at the University of British Columbia. He is a founding member of the Distributed Systems Research Group and director of the Open Systems Interconnect Lab. His interests include computer communications, distributed applications, and distributed operating systems. Currently he is working on the Raven project, an object-oriented distributed system. Neufeld received a BSc (with honors) and an MSc from the University of Manitoba and a PhD in computer science from the University of Waterloo in 1987.

The authors' address is University of British Columbia, Department of Computer Science, 2075 Wesbrook Mall, Vancouver, BC, Canada, V6T 1W5.

Introducing TurboCASE™, the fastest, most integrated Macintosh CASE tool.



```
DATA:
   input one
DEFINITION:

DESCRIPTION:

DOMAIN:
   [red | white | blue]
```

```
DATA:
   coin
DEFINITION:
   quarter + dime + nickel
```

```
Checking a sub-project fr
Error(s) in the DFD 'Data F
Error(s) in the STD 'State
Checking a sub-project fr
Error(s) in the DFD 'dfd':
Checking a sub-project fr
Error(s) in the DFD 'dfd1':
```

Input		Process	Ou
input one	event in	process 1	dat ou
red	ON	Enable	
white	ON	Disable	
blue	OFF	Trigger	
red	ON	Resume	

```
DOMAIN:
   [ON | OFF]
```

```
Error: two of the input rows
are the same. Please correct!
```

StructSoft, Inc., the developer of one of the top selling PC CASE tools, now marketed as Teamwork/PCSA™ by Cadre Technologies, Inc., is proudly announcing a second generation CASE tool, TurboCASE, for the Apple Macintosh.

TurboCASE is an integrated, multi-window, multi-methodology supporting CASE tool. It is extremely easy to learn and use. It supports Structured Analysis with or without the Real-Time extension. It will also support Data Modeling, Structured Design and Object Oriented Analysis and Design in the future.

TurboCASE generates ASCII information exchange formats which can be used to link with Teamwork and Iconix' PowerTool™

A demo diskette is available for $15.00.

StructSoft, Inc. 5416 156th Ave. SE, Bellevue, WA 98006. Tel: 206-644-9834 Fax: 206-644-7714

Trademarks: TurboCASE : StructSoft, Inc.; Teamwork, Teamwork/PCSA : Cadre Technologies, Inc.; PowerTool : Iconix.