A version of this paper appeared in Proceedings of the 14th International Conference on Distributed Computing Systems, Poznań, Poland, 1994

A Group Structuring Mechanism for a Distributed Object-oriented Language

Przemysław Pardyak and Brian N. Bershad
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195, USA

Abstract

This paper describes a structuring mechanism for grouping objects in a distributed object-oriented language. A group structuring mechanism provides a single flexible method for managing distributed applications that involve complicated communication protocols and sophisticated structure. We have added such a mechanism to the Emerald distributed object-oriented language and its runtime system. Our group structuring mechanism fits entirely within the context of object-oriented programming, so similar mechanisms could be added to other distributed object-oriented languages.

1 Introduction

In this paper we show that a general and flexible group structuring mechanism for expressing and managing clusters of cooperating objects can be introduced into a distributed object-oriented programming language. A group structuring mechanism can ease the complexity of programming distributed systems by encapsulating the structure and communication protocols for groups of interacting objects.

1.1 The need for a group mechanism

Many problems in programming distributed systems are caused by the inherent complexity of their structure and communication patterns. Distribution adds a new design dimension requiring a designer to take into account placement of objects, their mobility, reliability of nodes and links of the underlying network, and the impossibility of consensus among parts of a distributed system. Several techniques are used to deal with this complexity. Elaborate protocols are used to maintain (at least partial) consistency among components of a distributed system. Replication and recovery mechanisms are used to provide higher availability and fault-tolerance.

Group communication is another recognized technique for managing the complexity of distributed systems [2, 10]. It is used to simplify communication by enabling simultaneous addressing of all members of a group. It helps maintain consistency by ordering messages. Group communication protocols provide support for structuring techniques like replication and load sharing and may be used as a basis for fault tolerance [1].

Group communication protocols are themselves complex, difficult to write, and hard to manage. They require that the client programmer specify the properties of message ordering, policies for handling faults, and methods for interpreting a multiplicity of results following single invocations. Therefore a group communication mechanism alone is insufficient for expressing and managing the complexity that arises from arbitrary styles of group interaction. To demonstrate this, consider the use of a group communication mechanism in the context of a replicated service where a request to the service is turned into requests to each replica and some protocol is run to assure consistency among the replicas. A number of aspects have to be taken into account when designing such a system: number of replicas, placement, consistency mechanism and policy, failure handling, and protocols for handling client requests. Consider the problem of handling the results of group invocation when several servers are called as part of a multicast and each can return a result. The result handling protocols may vary from a simple one, such as ignoring all results or waiting for a fixed number of them, to more complex ones, e.g., collecting results from a majority of servers. In the most general case the number of collected results may depend on the number of servers, the parameters to the call, and the results already received. For most applications, a client of such a replicated service should see it as a simple service without being exposed to the complexity of its implementation.

A group structuring mechanism is a facility for creating and maintaining groups. By encapsulating group communication in such a mechanism, we can achieve a simple interface to a group without exposing the complexity of its structure and internal communication.

1.2 Goals

A group mechanism should be flexible to enable the specification of all aspects of group interactions [14, 13] and transparent to hide the complexity of these interactions. In general, we wish to support three degrees of transparency for clients of group communication:

completely transparent – the client is oblivious that it is interacting with a group; members of a group are not

This research was supported by Komitet Badań Naukowych under grant no. 8.0077.91.01 and by a National Science Foundation Presidential Young Investigator Award. Pardyak performed this work while at the University of Mining and Metallurgy, Cracow, Poland.

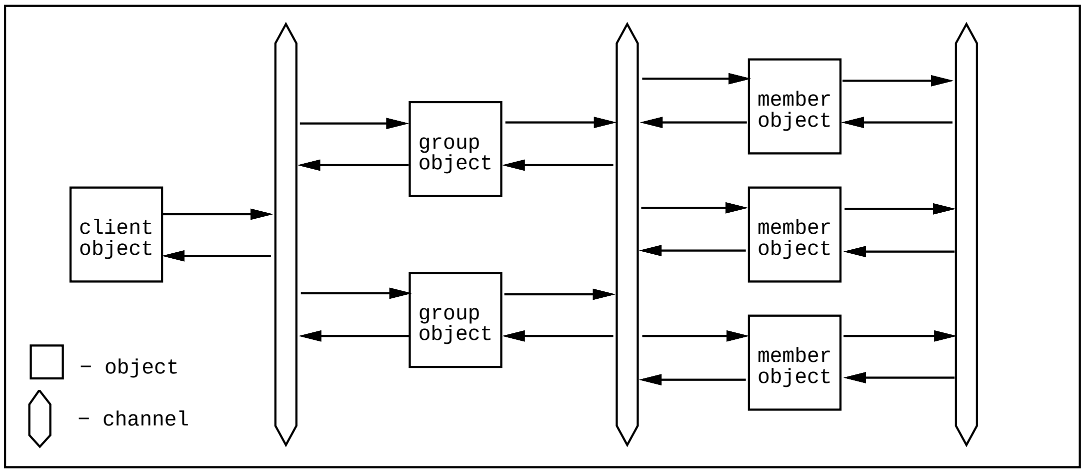

Figure 1: *General model of a group structure. Clients communicate with group objects via the group channel. Group objects interface clients and members via the group channel. Members are connected by the member channel.*

visible; results are collected by the group; a single result based on results collected from members is returned to the client.

semi-transparent – the client knows that it interacts with a group; group communication is controlled by parameters passed by the client; a single collected result is returned.

no transparency – the client knows that it interacts with a group; it controls execution of group communication; all results are returned separately.

The group structuring mechanism described in this paper allows varying degrees of group transparency to be expressed. For example, the client may not be aware that it is communicating with a group, it may be aware, but not care to modify its behavior, or it may be aware, but only care to deal with the group when, say, fewer than half of the group members are accessible. These and other constraints can be expressed entirely within the framework of an object-oriented system, so no new programming paradigm is required.

Successful distributed programming models have equivalents in traditional programming languages, for example, RPC [3] and distributed shared memory [11]. In this paper, we propose a set of cooperating objects as a programming technique that provides a structuring mechanism for using groups and group communication. The technique is expressed in terms of the object-oriented programming paradigm. We are using the technique in the Emerald object-oriented programming language.

1.3 The rest of the paper

The rest of the paper is organized as follows. In Section 2 we discuss the requirements for a flexible and transparent group mechanism. In Sections 3 through 6 we discuss our group structuring mechanism first in general terms, then in the context of Emerald, and finally in terms of implementation. In Section 7 we discuss related work. Finally, in Section 8 we present our conclusions.

2 Characteristics of groups

In this section we present our model of a group structure, discuss the different roles served by a group, and advocate encapsulation of the group structure and communication.

Structure of a group

In [14] we proposed a general model of a group structure. According to this model, a group consists of two kinds of objects: group objects that interface the group with the outside world, and member objects that perform the group's tasks (see Figure 1). Group objects identify a group and provide access to its services. They are the only interface between client objects and a group. A client's request for a service, passed to a group object, is turned into appropriate actions of member objects. Members may have different functionality and they do not have to respond to the same requests. Objects comprising a group and its clients communicate through channels which represent the communication media used in the group and may be either a communication mechanism used to connect objects or an object that encapsulates such a mechanism.

Different views of a group

A group plays three roles: it is maintained by some manager objects, it is accessed by its clients, and it interacts with its members. Each role gives a different perspective on the group. For manager objects, a group is a collection of objects with a number of operations to maintain it. Such a collection may be created, its structure and membership dynamically changed, and its actions may be controlled and modified. For a client, a group is a server that offers a certain service. The implementation of such a server,

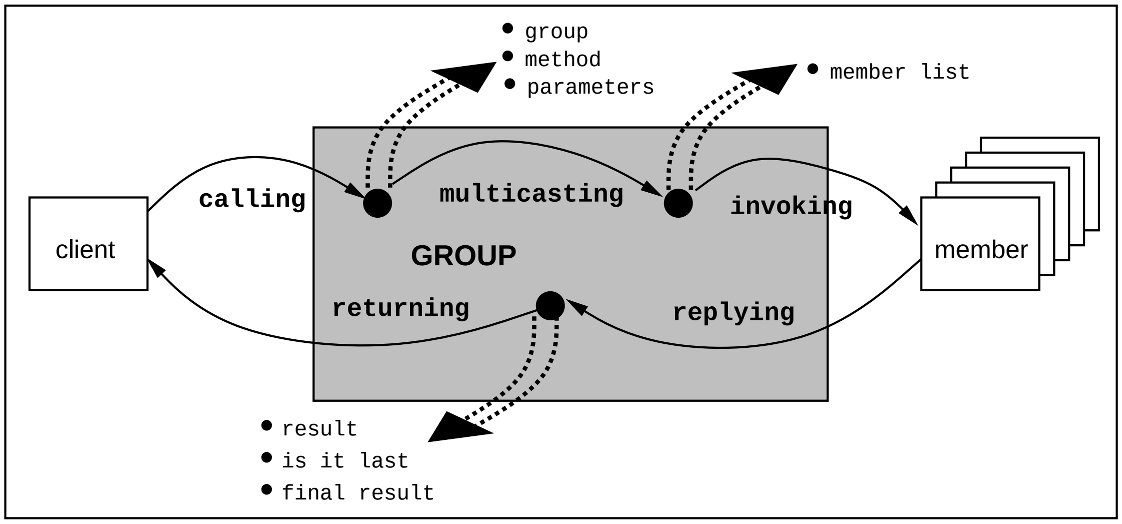

Figure 2: *A scenario of the MOI. The big rectangle denotes a group structuring mechanism. Arcs stand for steps of the MOI. The big arrows point to the information that becomes available at the end of each step.*

in this case the fact of it being a group, may be irrelevant to clients, so the their view of a group does not include managerial operations. For members, a group is a client that expects a certain functionality from them.

Encapsulation

A transparent group mechanism should hide two aspects of complexity in distributed systems: structure and communication. A group should be seen as an atomic, indivisible object, hiding its structure and the protocols for accessing members. A fully encapsulated group may transparently change the number, identity, and configuration of its members, as well as the protocols for passing a client's requests to the members. A client of such a group sees a simple interface despite the group's internal complexity.

3 A proposed mechanism

The design of our group mechanism is focused on the flexible and transparent handling of group communication. We first show the issues of group communication that affect the design and then present a group structure that takes these issues into account.

3.1 Group communication for objects

Multiple Object Invocation (MOI) [14] is a method of providing group communication among objects. It is based on the following scheme: an object identifying a group is invoked, the invocation is turned into invocations of the appropriate methods on appropriate member objects, and results of these invocations are collected and returned as the result of the original invocation. Each result is processed at the moment of its reception without waiting for the others. Multicast invocation may be abandoned by the caller at any moment if the results already collected are sufficient for it to proceed.

In Figure 2, we show a scenario of the MOI. In the first step (*calling*), a client's message is passed to the group. It carries information about the group being called, the invoked method, and arguments of the invocation. The next step (*multicasting*) encompasses protocols used for the selection of members to be called, message ordering, and synchronization. As a result of this step, a list of member objects to be invoked becomes available. The third step (*invoking*) consists of invocations of selected member objects. In the next step (*replying*), the invoked member objects return results of the invocations. Knowledge of the list of invoked members allows the group to detect if the returned result is the last one. When enough results have been collected, the reply to the client is determined and returned (the *returning* step).

An important feature of MOI is that it enables multicast communication among objects in such a way that both clients and members see the invocation as a traditional (client/server) object invocation. MOI does not require any changes to the client or server. Any method may be invoked either directly or by means of the MOI mechanism. Any object may invoke a group obliviously of group communication.

3.2 Group structure

In order to provide a means to flexibly handle the MOI, the group structure in our mechanism reflects steps of the MOI (Figure 2). The mechanism is a framework for maintaining the group's membership and handling group communication. A group may be created from the framework by providing a number of objects (group components) that share the responsibilities of group communication (Figure 3). Each of these components encapsulates a single step of MOI and is invoked by the framework when this step

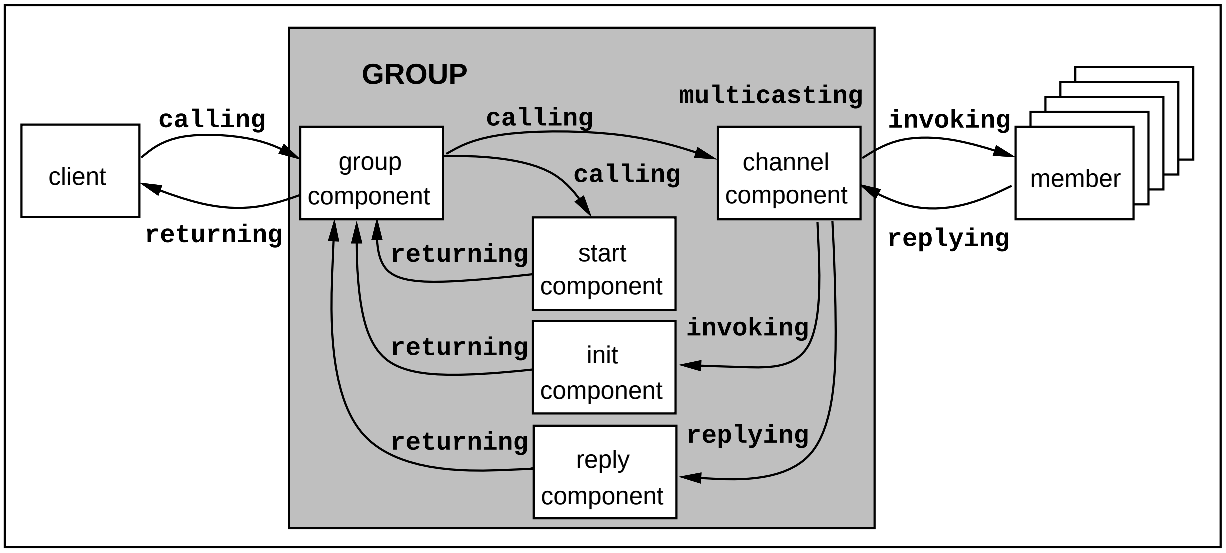

Figure 3: *Structure of a group in the proposed mechanism. Arrows depict the steps of the MOI (Figure 2). The group object according to Figure 1 is the group component here. The group channel is the channel component that encapsulates multicast of member invocations. The client channel is a direct invocation on the* group *object.*

should be performed. In this way, the framework enables group communication by linking the group's components and invoking them according to the execution of MOI. It does not define any particular protocol for handling group communication. This protocol is realized by the actions of component objects and is independent of the framework.

Some of the group's components may be provided by the same object. Different methods of the object may be used to provide functionality of more than one component.

The *Group* and *Channel* components perform actions that interface the group's clients and members. The *Group* component is the only object visible to the group's clients. Its converts their requests into group's actions. Eventually, results of the group's invocation are passed by the *Group* object back to the client. The *Channel* component performs all actions that lead to invocations of members, including selecting members to be invoked, multicasting, ordering of the invocation with respect to other invocations, and synchronizing the invocation with other events. The actions of the *Channel* component are not split into separate steps because some may be unnecessary in some cases and their order may differ from group to group. For example, a group may require selecting half of the members and ordering their invocation relative to all other group invocations in the system. Another group may need ordering relative to changes in its membership and then selecting only one member. The results of member invocation are passed back to the *Channel* component.

Three additional components are used by the group to dynamically react during the MOI. The *Reply* component encapsulates the policy of collecting results. It is passed each result separately and informed by the *Channel* component about which result is the last one. The *Reply* component decides on the final result of a group invocation and passes it to the *Group* component. The *Start* and *Init* components permit a group to undertake actions in case of the events during the MOI that mark a change of the state of invocation and which are independent of member invocations. The *Start* component is invoked when the MOI is initiated. The *Init* component is invoked at the time the *Channel* invokes members.

Additional infrastructure exists in a group for handling group membership, exceptions, and modifying group's structure and behavior.

A group has a dynamic structure. It can be reconfigured after creation by replacing or modifying the objects comprising it. By allowing more than one component of each kind, we provide a mechanism for creating groups with multiple personalities (a personality is a view of a group as seen by a client through the particular *Group* object it is accessing). We require that each of the group's personalities see the same set of members because it is the membership that defines a group. We assure this by requiring that all of the group's personalities share the infrastructure for membership handling.

3.3 Properties of the mechanism

The structure of a group enables flexible MOI protocols because knowledge about the course of group communication is made available to the application. The mechanism fully encapsulates both the structure of the group and its internal communication. Group communication is fully transparent because client and member objects interact with a single object by means of a simple invocation. Semi-transparent group communication may be achieved by modifying a group's behavior according to parameters passed to the *Group* object. Other levels of transparency

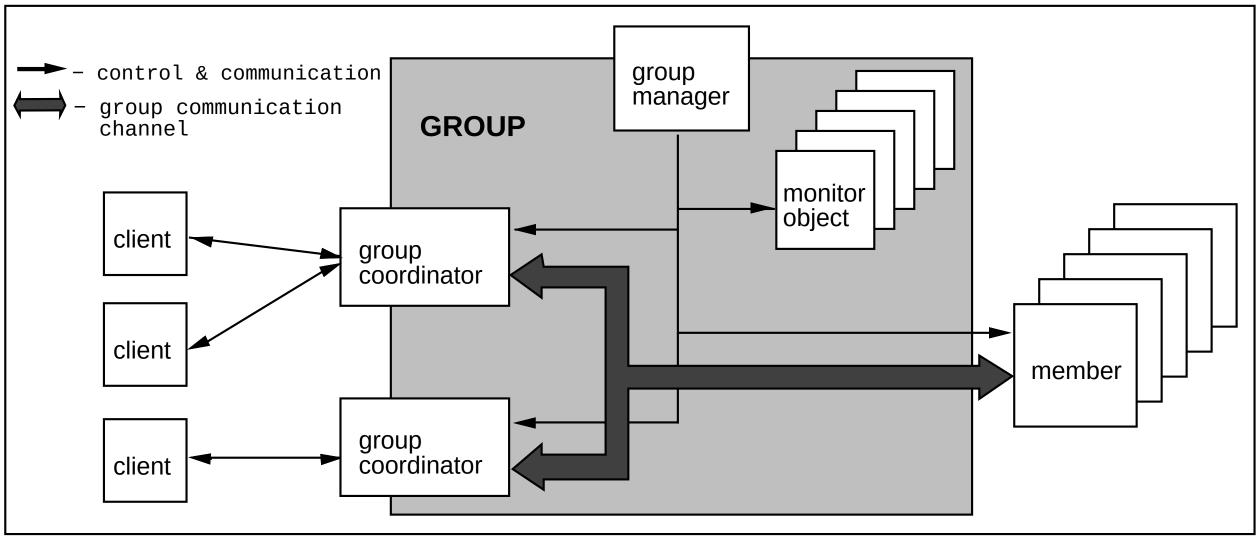

Figure 4: *Structure of a group in Emerald. A group is managed through its manager. Its services are accessible though its coordinators. Monitor objects are notified of group events such as failures and changes in membership.*

can be achieved by enabling interactions between clients and the internal group objects.

4 Adding groups to Emerald

In this section we describe the implementation of our group mechanisms in Emerald [15]. Emerald is an object-based programming language and a distributed runtime system for objects. The language is strongly typed and supports distribution. Emerald's objects are mobile and may be invoked in a location independent manner.

4.1 Group structure

The central objects in an Emerald group are group coordinators (Figure 4). A coordinator corresponds to a *Group* object in our model in that it is the client's handle to group services and it defines actions undertaken upon group invocation. A coordinator handles group communication and combines the functionality of the *Start*, *Init*, and *Reply* objects from Figure 3. The functionality of the *Channel* object is realized by the underlying run-time system and permits only a fixed number of predefined multicast protocols.

Two objects represent a group to outside objects: the group coordinator which corresponds to the client view of a group and provides access to services offered by a group to its clients, and the group manager which corresponds to the managerial view of a group and provides methods for maintaining the group. Since the coordinator defines groups services, a group may have multiple personalities by having multiple coordinators.

A group contains a set of members which respond to the same invocations although each may react differently. There are also secondary members of a group called monitors that monitor group events like membership changes and failures.

4.2 Group maintenance

A group is created in two stages. First, the *Group* built-in object is used to instantiate a group creator. Second, the group creator creates the actual group (Figure 5). Two abstract types are used to create a group: the type of member objects and the type of the service the group will provide. A group object conforms to the managerial view of the group and is used for maintaining its membership. Members and monitor objects may be added to and removed from a group and a group's membership may be inspected. To obtain the client's view of the group, the *GetGroup* method is invoked on the object. The object it returns (a group coordinator) fully encapsulates all group mechanisms, conforms to the service type of the group, and may be used wherever a normal object of the same type can be used.

An object that on request creates group coordinators is passed as a parameter of group creation. This object is invoked whenever a client asks for its handler. If a group should have only one personality, the object creates only once coordinator and returns it to all clients. If the object creates more coordinators the group has multiple personalities.

4.3 Group communication

A group coordinator may have traditional operations as well as **multicast** operations implementing an MOI protocol. If an operation invoked on a group is a traditional one then it is normally invoked on the coordinator object. If it is a **multicast** operation then it is turned into invocations on members according to the protocol specified in a **call** clause. The three parts of the operation body (**start**, **init**, and **reply**) specify reaction to subsequent events that occur during the MOI: starting an invocation, invocation of members, and reception of a result.

```
% creation of a group
NSCreator ← Group.Of[NSServiceType, NSServerType]
NSGroup ← NSCreator.Create[NSCoordinatorCreator]

% building a group
NSGroup.Join[Server1]
...
NSGroup.Join[Server9]
NSGroup.Leave[Server2]

NSGroup.AddMonitor[NSMonitor]

% client's view of a group
NameService ← NSGroup.GetGroup

% using a group
NameService.Add["sirpa", Sirpa13]
NameService.Remove["foo", FooBar]
foo ← NameService.LookUp["sirpa"]
```

Figure 5: *A sequence of operations for creation and use of an example group (a replicated name service).*

Figure 6 shows an example coordinator with four operations. The *Add*, *Remove*, and *Lookup* operations are group communication invocations. The first two use a totally ordered reliable multicast protocol to invoke all the group's members. The third one invokes only one member. The *Add* operation is aborted if the group has no members (**init** section). The **reply** section is called whenever a result is returned from an invoked member. The *GetServers* function is a traditional operation that returns a list of current group members by invoking the group object.

4.4 Communication protocols

Our group mechanism can use any protocol supported by the underlying system. In the Emerald implementation, we use a range of protocols including reliable broadcast protocols with different ordering properties (total, causal, total causal, and unordered) as well as singular calls (in a singular call one member is selected, if it cannot be reached another one is selected), and n-call (like singular call but invocation of n members is required).

Protocols to be used for group maintenance are defined in the coordinator creator. Protocols used for multicasting MOI invocations are defined in the **call** clause of the **multicast** operation in a group coordinator. All messages in the system are ordered relatively to each other.

4.5 Receiving broadcast invocations

An optional clause **by** has been added to the method definition syntax to enable detection of an MOI invocation and to reveal a list of all objects that received the same invocation message. In case of an MOI invocation of the method, the variable declared by the **by** clause contains a list of active members of a group, and a special object *NIL* otherwise. If the list of receivers is not needed then the **by** clause may be omitted—this means that the MOI mechanism is transparent on the callee's side. Objects initially not designed as parts of a group may be used in group invocations without any change.

```
coordinator NameServiceController
  % operations available on the group
  export Add, Remove, LookUp, GetServers
  % an operation turned into MOI on members
  multicast Add[Name : String, Item : Any]
    % use atomic causal broadcast protocol
    call Protocol.Total

    % called when members invoked
    init Receivers : Vector.Of[MemberType]
      % abort MOI when no members in the group
      quit when Receivers ==NIL
    end init

    % called when a member returns
    reply received
      last isLast : Boolean

      % stop when all members returned
      quit when isLast
    end reply
  end Add

  multicast Remove[Name : String, Item : Any]
    ...
  end Remove

  % an invocation called on a single member
  multicast LookUp[Name : String] → [Item : Any]
    % use singular broadcast (invoke one member)
    call Protocol.Singular

    % wait for the first result and return
    reply Result : Any received
      Item ← Result
    end reply
  end LookUp

  % operation performed locally on the coordinator
  function GetServers → [Res : Vector.Of[MemberType]]
    % call the associated group object
    Res ← manager.GetMembers
  end GetServers
end NameServiceController
```

Figure 6: *Example of a group coordinator. This object encapsulates group communication protocols used by the replicated name service.*

5 Implementation

Implementation of our group mechanisms consists of three major parts: broadcast protocols implemented in the Emerald kernel, support for groups of lightweight objects in the object run-time system, and group communication primitives in the language and run-time system.

The Emerald kernel was augmented with a suite of reliable broadcast protocols. The suite is based on reliable broadcast and contains a number of ordering protocols (unordered, total, causal, total causal) and a protocol for detecting and recovering from inconsistencies in the system.

The protocols use physical broadcast over Ethernet using UDP. The consistency protocol enables nodes to be added and removed from the system, and to recover from node crashes and transient lapses of connectivity.

The object group mechanism uses broadcast to propagate group membership information to all nodes of a system. Each node maintains a list of members for all active groups. When a group initiates a broadcast, the invocation message is broadcast to all nodes. Upon receiving a broadcast message, a node invokes all local group members and subsequently sends back their replies. The Emerald object migration mechanism propagates broadcast and membership information when members or parts of a group migrate.

The changes to the Emerald language include a new built-in object (*Group*) and a number of group communication primitives (the **call**, **start**, and **reply** sections of the **multicast** operation, and the **by** clause). The changes do not compromise the Emerald object model and are typesafe.

6 Performance

In this section, we present the costs of some group communication and maintenance operations measured for a distributed name service. The service consists of name servers placed one on each node of the system and creating a group. The measurements were taken on a system consisting of 5 nodes (40Mhz Sparc IPX) connected by 10Mb/s Ethernet. MOI and group maintenance operations used total causal broadcast.

First, we measure group maintenance cost. Creation of a distributed service based on a group requires two steps: first, the creation of all objects comprising it and their distribution, and second, construction of the group (creation of the group and addition of its members). In our example application, the first step took 135 ms and the second step 42 ms. The cost of group related operations is less then 25% of the total cost of creation of the service (177ms) which means that they do not have a prohibitively high cost.

Second, we contrast times of different invocation methods by measuring the time to invoke the empty method by means of MOI and remote object invocation. To show the impact of collecting results, we measure a broadcast invocation that is abandoned without waiting for results and an invocation that collects all results. The asynchronous MOI (10ms) is significantly faster than a remote invocation of an object (30 ms). The cost of fully synchronous MOI (51 ms) is of the same order of magnitude as the cost of remote invocation. Table 1 summarizes the measurements.

operation	time
creation and distribution of servers	135 ms
construction of a group	42 ms
total time of creation of the service	177 ms
remote invocation of a single object	30 ms
MOI, abandon results	10 ms
MOI, collecting all results	51 ms

Table 1: *Measurements of basic group operations.*

7 Related work

Group communication has been recognized as a valuable communication method for distributed operating systems. Systems like V [5] and Amoeba [10] use process groups for replicated and distributed services. Multicast is used to simplify and expedite communication for process groups. The ISIS reliable distributed toolkit [2], also based on process groups and multicast, enables group communication to increase fault-tolerance. Remote Procedure Call has been enhanced to handle communication with multiple processes simultaneously [16] and to support replication [6].

The success of group communication in distributed systems provoked research into adopting the concept as a programming technique. Broadcasting Sequential Processes [7] enabled sequential processes similar to those in Hoare's CSP [9] to broadcast messages to all others although there was no support for groups or transparency (messages were explicitly broadcast and received). Another approach relies on class libraries to abstract the group communication primitives of the underlying operating system [8, 12]. These libraries are not a general language construct because their semantics depends on the system they are based on. They are not transparent because multiple responses to an invocation must be explicitly handled by a client.

We are aware of one other effort to add group mechanisms to Emerald [4]. Their approach supports a simple form of group structuring, whereby new members can be added to an existing group. Group invocation results in one member of the group receiving the invocation. Otherwise, there is no support for group communication. Our mechanisms allow us to emulate this behavior using singular invocations.

8 Conclusions

In this paper, we have presented a group structuring mechanism for a distributed object-oriented language. We have shown that the notion of a group fits well into the paradigm of object-oriented programming. A group may consist of objects and may itself be viewed as an object. Group communication may be expressed by means of object invocations. Groups and group communication abstract issues such as communication protocols, membership, and ordering of messages.

Acknowledgements

We would like to thank Grzegorz Czajkowski and Krzysztof Zieliński from the University of Mining and Metallurgy in Cracow and Bjarne Steensgaard from Microsoft Research for their comments on earlier versions of this paper.

References

[1] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[2] K. P. Birman and R. Cooper. The ISIS Project: Real Experience with a Fault Tolerant Programming System. *Operating Systems Review*, 25(2):103–107, April 1991.

[3] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] A. P. Black and M. P. Immel. Encapsulating Plurality. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 57–79, July 1993.

[5] D. R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, May 1985.

[6] E. C. Cooper. Replicated Procedure Call. *Operating Systems Review*, 20(1):44–56, January 1986.

[7] N. M. Gehani. Broadcasting Sequential Processes (BSP). *IEEE Transactions on Software Engineering*, SE-10(4):343–351, July 1984.

[8] O. Hagsand, H. Herzog, K. Birman, and R. Cooper. Object-oriented Reliable Distributed Programming. In *Proceedings of the 2nd International Workshop on Object-Orientation in Operating Systems IWOOOS'92*, Dourdan, France, September 1992.

[9] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[10] M. F. Kaashoek and A. S. Tanenbaum. Group Communication in the Amoeba Distributed Operating System. In *Proceedings of the 11th International Conference on Distributed Computer Systems*, pages 222–230, Arlington, VA, May 1991.

[11] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 229–239, August 1986.

[12] M. Makpangou, Y. Gourhant, and M. Shapiro. BOAR: A Library of Fragmented Object Types for Distributed Abstracitons. In *Proceedings of the International Workshop on Object Orientation in Operating Systems IWOOOS'91*, pages 164–172, October 1991.

[13] P. Pardyak. Group Communication in an Object-based Distributed System. Master's thesis, University of Mining and Metallurgy, Cracow, Poland, 1992.

[14] P. Pardyak. Group Communication in an Object-Based Environment. In *Proceedings of the 2nd International Workshop on Object Orientation in Operating Systems IWOOOS'92*, pages 106–116, September 1992.

[15] R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A General-Purpose Programming Language. *Software — Practice and Experience*, 21(1):91–118, January 1991.

[16] M. Satanarayanan and E. H. Siegel. Parallel Communication in a Large Distributed Environment. *IEEE Transactions on Computers*, 39(3):328–348, March 1990.