Dept. of Computer Science, Cornell University, December 1991.

[5] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object Structure in the Emerald System. In *Proceedings of the 1986 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–86, October 1986.

[6] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

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

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

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

[10] H.-M. Jarvinen, R. Kurki-Suonio, M. Sakkinen, and K. Systa. Object-Oriented Specification of Reactive Systems. In *Proceedings of the 12th International Conference on Software Engineering*, pages 63–71, 1990.

[11] M. Frans Kaashoek and Andrew 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.

[12] Luping Liang, Samuel T. Chanson, and Gerald W. Neufeld. Process Groups and Group Communication: Classification and Requirements. *Computer*, 23(2):56–66, February 1990.

[13] Mesaac Makpangou, Yvon Gourhant, and Mark Shapiro. BOAR: A Library of Fragmented Object Types for Distributed Abstracitons. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*, pages 164–172, October 1991.

[14] P.M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast Protocols for Distributed Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[15] Rajendra K. Raj, Ewan D. Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A General-Purpose Programming Language. *Software — Practice and Experience*, 21(1):91–118, January 1991.

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

```
operation Bind[Name : String, Item : Any]
   by Receivers : Array.Of[Any]
   if Receivers == NIL then
      % simple invocation
   else
      % broadcast invocation
   end if
end Bind
```

Figure 8: Example of a use of the **by** clause

- all members of a group receive all messages.
- each member of a group sees the same sequence of events related to the group.
- if two object belong to the same two groups then they see events associated with the same group in the same order but the relative ordering of events belonging to both groups may be different.
- any object (in particular members of different groups) sees events initiated by the process of the same object in the same sequence .

Although channels provide only one level of synchronism, future development of Emerald should encompass all possible levels We plan to change the channel creators to have methods like *CreateAtomic* and *CreateCausal* instead of only one *Create*.

4.6 Use of channels

All the changes to the language form a consistent set of tools, that enable a programmer to specify group actions according to the model described in the previous sections. Groups may by joined and left on-the-fly without restriction. The same methods may be used in both traditional and group invocation. A list of receivers is available both at the caller and callee's side. Just like traditional invocation in Emerald, group communication is type safe and is orthogonal to mobility and distribution.

In contrary to Fragmented Objects, Emerald group mechanism is general i.e. it does not assume a specific architecture of objects (fragmentation) and may be used together with mobility.

5 Summary

We have devised and described a general model of a group structure. Multiple Object Invocation has been proposed as a method for performing group communication among objects. We have implemented MOI in Emerald object-oriented language and operating system.

The general model may be used for structuring communicating modules into hierarchical groups. Multiple Object Invocation and its Emerald implementation — channels — accomplish group communication in an object-based environment. They allow to invoke concurrently any number of objects simplifying the communication pattern and leading to increase of parallelism. A flexible mechanism of result handling provides for a necessary level of asynchronism which may be crucial in case of large groups. As invocation of a member object may be done indirectly by the group's identifier, communication with unknown objects is feasible.

Multiple Object Invocation is a mean of specifying group communication in an object-oriented environment. In spite of its advantages, group communication is only the first step towards the ultimate goal of the author's research: a general method of programming with groups.

Acknowledgements

I would like to thank my supervisor Eric Jul for his help in writing this paper.

References

[1] M. Aspnas, R.J.R. Back, and R. Kurki-Suonio. Efficient Implementation of Multi-process Handshaking on Broadcast Networks. Reports on Computer Science A75, Åbo Akademi University, Turku, Finland, March 1989.

[2] M. Stella Atkins, Garnik Haftevani, and Wo Shun Luk. An Efficient Kernel-level Dependable Multicast Protocol for Distributed Systems. In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pages 94–101, October 1989.

[3] Kenneth Birman, André Schiper, and Pat Stephenson. Lightweigth Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[4] Kenneth P. Birman, Robert Cooper, and Barry Gleeson. Design Alternatives for Process Group Membership and Multicast. Technical Report TR-91-1257,

4.2 Broadcast statement

The purpose of the **broadcast** statement is to perform Multiple Object Invocation, i.e., invoking all objects constituting a group and subsequently collecting the results of these invocations. An example **broadcast** statement may be seen in Figure 7. The **broadcast** statement consists of:

- a **broadcast** clause
- a **to** clause
- an initial process
- a **reply received** section

The **broadcast** clause gives the information about a group being invoked (a *NameChannel* placed after the **on** keyword) and the message (*LookUp*[]) being passed to it. All objects that subscribe to this channel, are concurrently invoked with the message. When the invocation has been received by all member objects, a constant declared by the **to** clause is initialized to a list of receivers and the initial process of the **broadcast** statement is started. The initial process may include declarations of variables used inside of the statement and a sequence of statements which are executed as a preparation for receiving results of the invocation. Typically it would depend on the list of receivers, for instance a number of receivers may be checked and the statement aborted in case when no object received the message i.e. if the group is empty. The last part of the **broadcast** statement is the **reply received** section which allows collecting results of the broadcast invocation.

A **broadcast** statement may be terminated by executing a **quit** statement inside of it. All subsequent replies to the broadcast invocation are discarded. The processes inside of the **broadcast** statement continue their execution.

4.3 Handling results

As a result of initiating the **broadcast** statement a number of processes is started asynchronously to perform the invoked method. When the processes finish then the thread of control is returned to the **reply received** section in the broadcasting object. If the **broadcast** statement is placed in a monitor then the returning processes are executed sequentially, otherwise they run concurrently.

A **reply received** section consists of declaration of variables that contain results of the invocation (*Item* in Figure 7), an optional **last** clause, and a block of the section. A variable defined by the **last** clause indicates whether the received result is the last expected one. When a result is received than the variables declared by in the **reply received** and **last** clauses are initialized and the block of the section is executed.

```
function LookUp[Name : String] → [R : Any]
   broadcast LookUp[Name] on NameChannel
      to Receivers : Array.Of[ServerType]
      R ← NIL
      quit when Receivers.Empty
      reply Item : Any received
         last isLast : Boolean
         if Item !== NIL then
            R ← Item
            quit
         end if
         quit when isLast
      end received
   end broadcast
end LookUp
```

Figure 7: Example of a **broadcast** statement

4.4 Receiving broadcast invocation

The same method of an object may be either invoked directly on the object or it may be invoked as a part of Multiple Object Invocation performed on a channel the object subscribed to. An optional clause **by** has been added to a method definition syntax (Figure 4.4) to enable detection of multiple invocation and to reveal information otherwise unavailable at call time. A variable declared in this clause contains a list of all objects which received the same invocation message. If the invocation was not broadcast then the variable references the *NIL* object. If the list of receivers is unnecessary then the **by** clause may be omitted—this means that MOI mechanism is transparent on the callee's side: object initially not designed as parts of a group may be used in group invocations without any change.

4.5 Synchronism

In the current implementation of MOI, the level of synchronism is fixed. Group communication is performed with preserving atomicity, ordering is preserved only inside one group, causality is defined by a sequence of invocations initiated by the process of the same object. This means that:

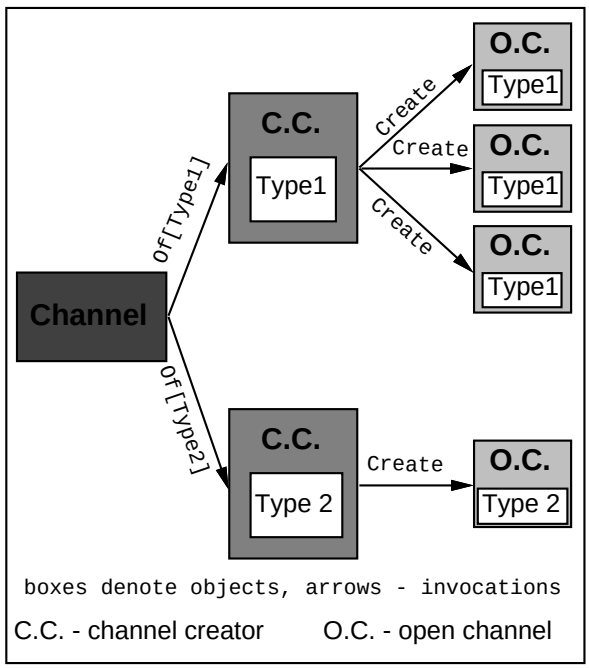

Figure 5: Channel creation

viewed as a run-time system for the language and language level objects are the first-class citizens of the system. The language is strongly typed and supports distribution. Emerald's objects are mobile and may be invoked in a location independent manner. Objects may be active; concurrency control is achieved by monitored sections inside objects [5]. The language provides for a number of built-in objects which are implementations of basic types (like *Integer* or *Array*) or abstract some system level features (e.g., a *Node* object representing an Emerald kernel).

Several enhancements have been added to the Emerald programming language to provide for tools for group communication by means of Multiple Object Invocation. The core of the Emerald implementation of group mechanism are channel objects created by a new built-in object *Channel*. They enable creation and maintenance of a group and may be used to perform Multiple Object Invocation on all objects belonging to it. Syntax of the Emerald language have been enhanced by the **broadcast** statement and the **by** clause. The first one is used to trigger a multiple invocation while the latter one enables a control of the invocation on the callee's side. All the changes to the language do not change the Emerald's object model. Group communication is type secure and supports mobility and distribution.

```
NameChannel ← Channel.Of[ServerType].Create
% building a group
NameChannel.Subscribe[Server1]
...
NameChannel.Subscribe[Server9]
% communicating with the group
broadcast GetServer on NameChannel
    ...
end broadcast
```

Figure 6: Example of operations on channels

4.1 Channel objects

The core of the Emerald's implementation of the group mechanism is the *Channel* object. It is a language level abstraction of the group communication primitives available in the operating system. These primitives include group creation, joining and leaving a group, and broadcasting an invocation to all group members.

A group is created in two steps in Emerald (see Figure 5). First a **channel creator** is created by a method *Of* of the *Channel* object. Second, an **open channel** is created by the *Create* method of an appropriate channel creator. An open channel is an identifier of a new group.

Two step creation of an open channel is necessary for making group communication based on channels type safe. Each open channel has a type associated with it which is a type of objects which may join the group identified by the channel. It is a type passed as an argument to the invocation of the *Of* method which returned the open channel's creator. Open channels returned by different invocations of the *Create* method are distinct objects and identify different groups but have the same type associated with them.

The *Subscribe* and *UnSubscribe* methods are used for maintaining a member list of a group (Figure 6). An object that is passed as an argument of the *Subscribe* method on an open channel becomes a member of the group identified by the channel and subsequently receives all invocation messages broadcast on it. The *UnSubscribe* method works likewise. If need be, a list of members of a group may be obtained by calling the *GetMembers* function of an open channel.

3.4 Failure handling

In the traditional object invocation, a failure of the invoked object becomes a failure of the invocation. In the Multiple Object Invocation this does not have to be the case: failure of a member object may be detected and recovered from and the result of the whole invocation may be based on the remaining results. Typically, when a failure of a single object occurs then:

- it may cause failure of the whole invocation.
- the failing object may be treated as it was never invoked.
- recovery action may be performed which returns a substitutional result.

A mechanism of Multiple Object Invocation should not exclude any possible method of failure handling. Therefore, no explicit assumptions about failures are included in its definition except that collecting of results should proceed in spite of failures of single member objects. A particular implementation may be supplemented with time-outs, ways of explicit signalling detected failures, or exception handlers.

3.5 Call-time information

Any method may be invoked either directly on the object implementing it or by the Multiple Object Invocation mechanism. Although the group mechanism may be transparent from the callee's side it may be important to reveal some information about the nature of the call. In the traditional invocation all information about it is available at the call time: the identity of a caller and a callee, the method's name and parameters. In contrast, a flexible mechanism of group communication should allow communication with an unknown number of unknown objects. If the membership of a group may be changed in parallel to invocations then information about the number and identity of members of a group becomes available at the call time. As this information may be critically important (e.g., algorithms that are based on accurate information about membership of a group) it should be revealed somehow to the receiver of a Multiple Object Invocation.

3.6 Group services based on MOI

Two levels of abstraction may be found in a specification of group services: group communication and group actions. The first is merely a mean of information exchange. In the case of Multiple Object Invocation this is multicasting of the invocation message and subsequent returning results. In contrary to that, group actions are all operations performed by a number of cooperating objects to fulfill their tasks. This encompasses both computations and information exchange, possibly in many distinct acts of communication involving different objects.

Multiple Object Invocation is a method of communication and therefore it cannot be thought of as a mechanism for describing group actions. It is, however, a building block of such actions: they may be constructed out of groups used in the MOI mechanism. The Multiple Object Invocation mechanism supported with a mechanism of group creation becomes a method for specification of raw groups. Such a raw group is a collection of objects which perform the same tasks by taking part in the same multicast invocations. Group services may be implemented by a single raw group but more complex groups may consist of several, possibly overlapping, raw groups.

The two level specification of group services leads to a composition of a group according to the general model from the previous section. Raw group objects, together with specification of multiple invocations they provide, become channels among those objects whose operations require group communication.

3.7 Features of MOI

The Multiple Object Invocation is proposed as a mechanism of group communication in object based environment. It is based on the assumption that a group should be identified by an object and that group invocation should be turned into traditional invocations on member objects. The MOI groups are created according to a simplified model of a group structure. Groups structured according to the general model may be constructed using MOI groups as building blocks. MOI is flexible due to asynchrony in handling results and possibility of aborting the invocation at any moment. Multiple Object Invocation may be therefore used in all applications which require a flexible group mechanism. Emerald channels, presented in the next section, are one of possible implementations of MOI.

4 MOI in Emerald

In Emerald [15, 6] the operating system and the language are tightly intertwined: the system may be

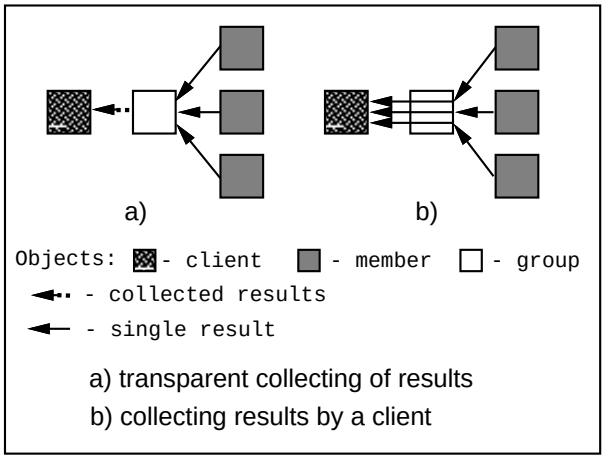

Figure 4: Collecting results of MOI

all the results. The reason for that is an unnecessary decrease of performance and parallelism: if an object is interested in any one of the replies out of a hundred then the first one received can do and waiting for the last hundredth one just to discard it is an obvious waste.

In general, any number of results of a broadcast invocation can be accepted: all, none, and everything in between. At the discretion of the programmer, the number of accepted results may be fixed either before the broadcast is initiated, at the time of broadcasting (e.g., may depend on the number or identity of receivers of the message), or may depend on the course of the invocation (failure detection, contents of results, time-out). The following examples of typical protocols for handling results of multicast illustrate the need for flexible method of handling results.

- ignore all results – full asynchrony is achieved.
- first k results – $k = 1$ is often used in case of replicated servers.
- given fraction of results – the results from a subset (e.g., majority or all) of the invoked objects are collected.
- results from non-failing objects – in case of failures some results may never be received and thus should not be waited for.
- first non-trivial result – example of application in a name service: a broadcast query about a given name should return the first match, but should detect the situation when a name cannot be resolved.

The conclusion which may be drawn from the above protocols is that any restriction put on result handling may cause a severe drop in performance and reduction of parallelism. Of course, any such protocol could be simulated by collecting all results and then performing selection but this would involve an unnecessary overhead. Because of that, a flexible solution is necessary in which an object which triggers a Multiple Object Invocation receives each result separately, possibly in parallel, and should be able to abandon the invocation at any time discarding replies not received yet. In the Multiple Object Invocation paradigm, a protocol of collecting results is not a part of the mechanism but of a specification; this approach leaves maximum flexibility for the programmer.

3.3 Group invocation

The results which are returned from objects invoked by Multiple Object Invocation may be collected either by a group object or by an initiator of the invocation (see Figure 4). In the first case, specification of result handling becomes a part of a group specification and a protocol for collecting them is constant, possibly parameterized by arguments of the invocation. In the second case, a method for gathering results becomes a part of a specification of an invocation at the client's side which leaves more flexibility but sacrifices the transparency of the group mechanism.

It is possible to implement both cases in the Multiple Object Invocation paradigm. If collecting protocols are placed outside of a group object then the object becomes a simple dispatcher of messages between clients and members of the group it identifies. The alternative solution makes the protocols an integral part of the group object's specification achieving transparency at the cost of flexibility. It is impossible, for example, to have two different protocols for handling results of the same method. If, for instance, all members have a method which merely returns a number (e.g., as a from of identification) then if once we want to get this number from all members and once only from one of them then either the method has to be duplicated or two groups must be created.

Generally speaking, if the more flexible solution is selected then a group is seen as a troupe of cooperating objects while in the other case it may be viewed as a single object transparently composing its actions from its member objects' actions.

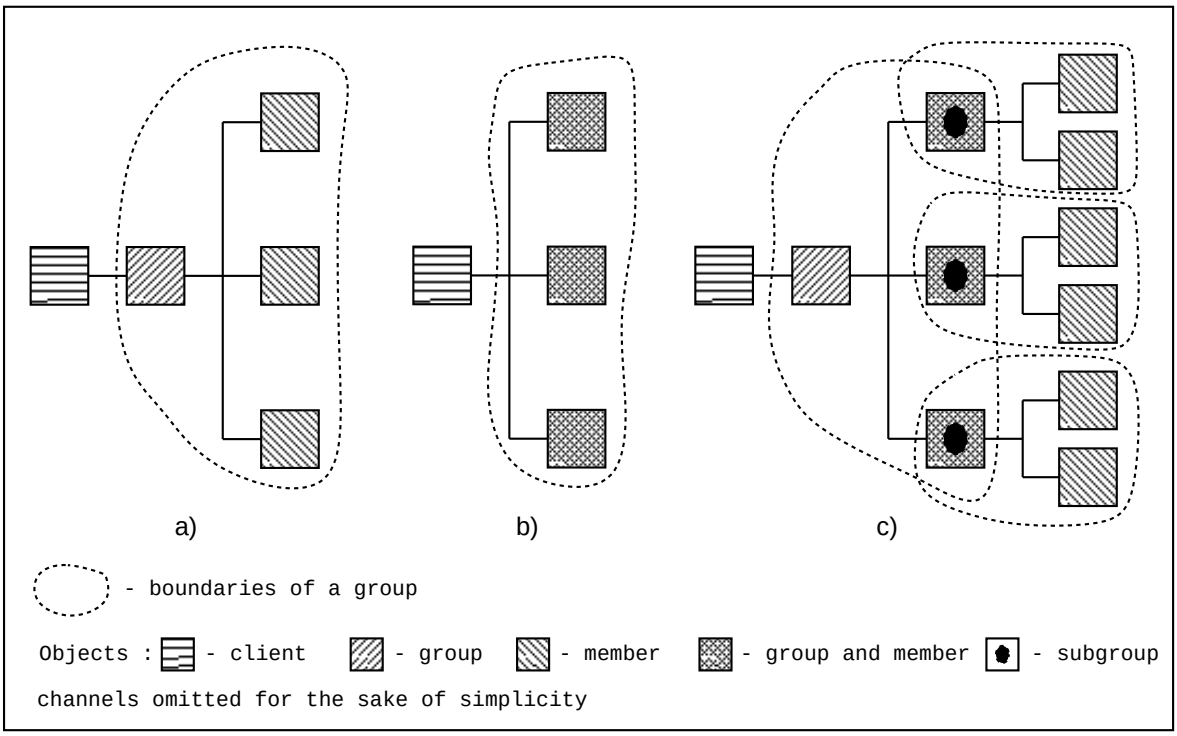

Figure 3: Examples of a group structure

the general one is that there is only one group object which is used as an identifier of a group in the multiple invocation. A channel between a client and a group object is a traditional object invocation. The Multiple Object Invocation mechanism is used as a channel between a group object and member objects. The interface between a group object and all the member objects is the same—it determines the methods which may be invoked on the group object and broadcast to all member objects.

A Multiple Object Invocation, just like a traditional invocation, consists of three steps: sending an invocation message, performing the invocation and collecting the results. The only difference between the two types of invocation is that, in the latter, an invocation message triggers execution of the appropriate methods in a number of objects instead of invoking one object. This leads to forking of new threads of control, one per invoked object. The processes created in the invoked objects proceed asynchronously and this asynchrony may be preserved in handling results: performing invocations and collecting results may overlap. Each result is processed at the moment of its reception without waiting for the others. Additionally, multicast invocation may be abandoned at any moment. If the already collected results are enough then the remaining ones may be ignored.

Summarizing, the Multiple Object Invocation is performed in the following way:

- caller invokes a group object
- the invocation message is broadcast to all members of the group
- all the objects are invoked; they execute the appropriate method
- results of individual invocations are returned and handled separately
- when enough results have been collected, the invocation is finished

3.2 Collecting results

When a number of objects have been invoked in a multicast invocation then a number of results (one per invoked object) is expected. An object invoking a group of objects may be interested in any subset of the expected results and should not be bound to collect

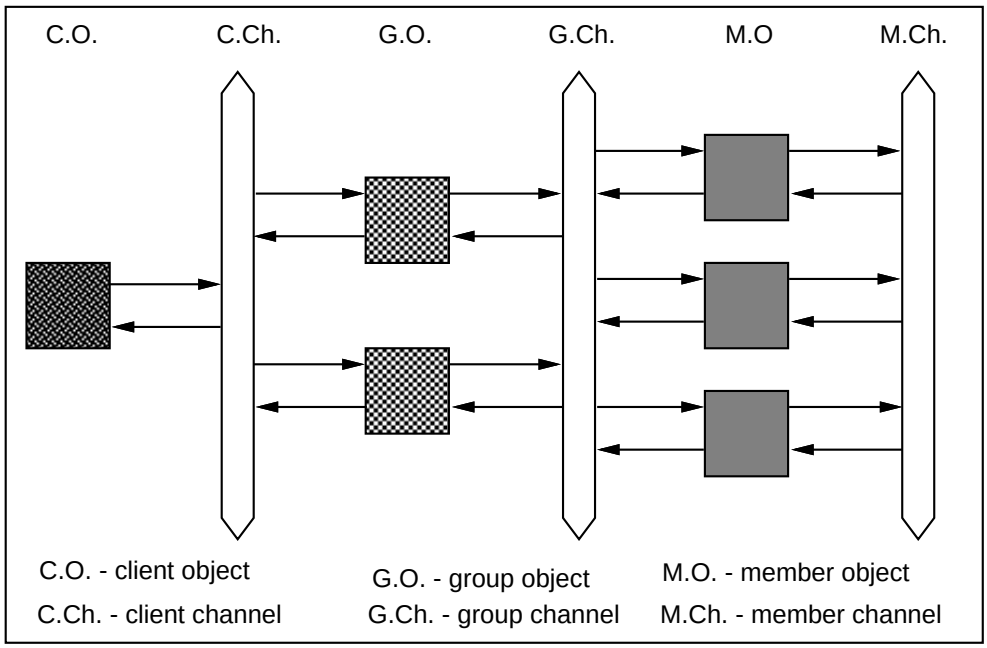

Figure 2: General model of a group structure

interconnection. Description of an interface describes a method of communication applied (e.g., RPC), its specification (types of arguments, etc.) and a protocol used (what sequences of calls are used for achieving which goals). Additional information is a method and level of synchronization used: communication with group may be done with preservation of atomicity (all addressees receive messages or none of them), ordering (each member receives messages in the same order), causality (causal dependencies among events are preserved). A group may also be synchronized with another one.

A group may be either static or dynamic, depending on whether its membership changes or not. A membership change may be associated with additional operations because of a need for preserving consistency. Additional information about a group's behaviour describes its operations upon failures. A group may, for instance, declare its failure, recover from it transparently, or it may reduce its functionality or performance.

3 Multiple Object Invocation

The general model presented in the previous section may be applied successfully for creating group services in an object-based environment. To achieve this, Multiple Object Invocation is proposed. It is based on a simplified model of a group and accomplishes group communication in it. Groups based on the general model may be constructed by combining groups based on the simplified model into more complex sets of objects as described at the end of the section.

As mentioned in the introduction, objects are the only entities in an object-based environment and the only mean of object interaction is method invocation. This should also apply after the introduction of groups into the objects' universe, so a group should be represented by an object and group communication should have a form of object invocation. In short, the idea of Multiple Object Invocation, which preserves this assumption, is based on the following scheme: an object identifying a group is invoked, this invocation is turned into invocations of the appropriate methods on all member objects, results of these invocations are collected and returned as the result of the original invocation.

3.1 Course of invocation

A simplified model of a group structure used in the Multiple Object Invocation is presented in Figure 3 (a). The main restriction in this model compared to

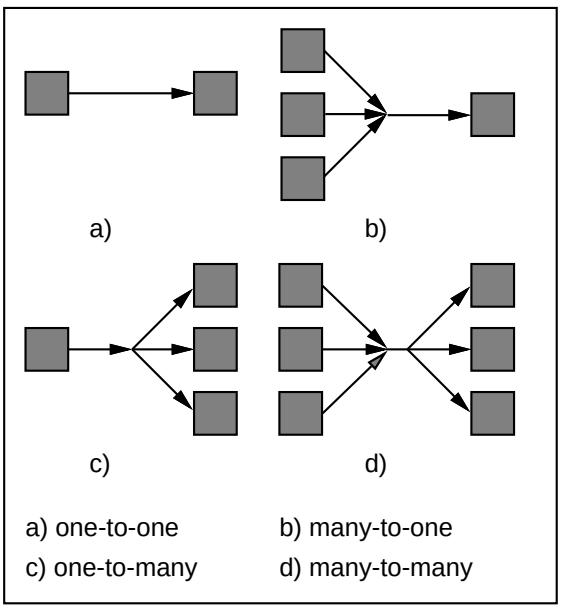

Figure 1: Methods of message passing

execute in parallel a number of procedures. An example of such concurrent procedure invocation may be MultiRPC or Multiple Object Invocation presented in this paper.

2.2 Overall group structure

The proposed model of a group structure is based on functionality of its components. According to the model, a group consists of objects which interface it with the outside world, objects which perform the group's tasks, and links among them. The objects belonging to the first class are called group objects. They identify a group and provide access to its services. The objects which actually accomplish the group's tasks are called member objects. A client's request for a service, passed to a group object, is turned into appropriate actions of member objects. All objects comprising a group may be linked by channels. A channel is a name for any communication mean applied in the group. It may be, for example, a simple object invocation or a Unix pipe. Figure 2 presents a structure of a group according to the general model.

Group objects are the only interface between client objects and a group. A group object makes it possible to view the group as a single object and allows clients to abstract from its implementation The role of a group object is manifold. It accepts clients' requests and may verify their legality and/or correctness. It dispatches messages from and to members. It may supervise the completion of a task. The recovery of the group may be initiated and controlled by the group object.

There are two kinds of activities performed by **member objects**. The first is associated with group configuration maintenance and may include such step as joining a group, leaving it, handling failures and reacting dynamically to changes of the configuration. The second is taking the responsibility for carrying out the group's tasks. The responsibilities and methods of fulfilling them may differ form member to member.

Channels are an abstraction of the form of communication among objects creating a group. There are three channels in a group: a client channel joining client and group objects, a group channel shared by group and member objects, and a member channel interconnecting member objects. A channel represents a mechanism used for communication among objects it connects. It does not have to be a single entity, e.g., a member channel may be a network of pipes fully interconnecting the set of members. A single channel may also comprise different methods, e.g., a group object may communicate with some members by means of RPC and with others by message multicast.

Some parts of the general group structure may be omitted in a particular group, e.g., a member channel may be not necessary, if member objects do not communicate with each other directly. A group may consist only of member objects, i.e., member objects may be also group objects and be accessible directly by clients.

Three examples of possible structures of a group are presented in Figure 3. In the first one a single group object passes clients' requests to members which do not communicate with each other. In the second example, group objects are identified with member objects. The third group is a hierarchical one: member objects of the main group are group objects of other groups.

2.3 Specification of a group

Specification of a group consists of the following items:

- group configuration
- interfaces associated with all channels
- protocol for maintaining membership
- protocol for exception handling

Configuration of a group is defined by a specification of objects which are included in it and their

On the other hand, group communication suits well collective tasks as it gives a method of describing the implementation of such a task in a more natural way due to the fact that it speaks in terms of sets of objects. In general, group communication in operating systems simplifies interprocess communication, adds to efficiency of a system, helps preserving consistency, and increases reliability.

1.3 Previous work

Communication based on multicast (broadcast) of messages has received much attention in the past few years. A variety of broadcast/multicast protocols [3, 14] have been implemented and group communication based on those protocols has been made available in some distributed operating systems [7, 11]. The requirements for group communication systems and their classification have been investigated in [12]. Some ideas on group communication semantics and on group construction may be found in [4].

The need for new programming paradigms using broadcast communication has been identified in 1983 [9] where Hoare's CSP have been augmented with capability of broadcasting messages. Since then a number of approaches have been designed. Message multicast is applied [2] in client – server troupe model. Multiple process rendezvous [1] enable synchronization of an arbitrary number of processes. MultiRPC [16] is an incorporation of multicast communication into an RPC-based distributed sequential programming. Another method of group communication based on RPCs is Replicated RPC [8] aimed at programming replicated services. A novel approach to specification of object-oriented reactive systems based on multi-object actions has also been devised [10].

Fragmented Objects (FO) [13] are aimed at supporting abstractions of distribution and provide a library of objects which may be applied by programmers for creating distributed applications. One of predefined Fragmented Objects are multi-point channels which may be used for multi-invocation of objects bound to them. The FOs, however, is more a toolkit for managing distribution than a general mechanism for programming with groups.

1.4 A proposal

Our proposal consists of two parts. First, a general model of a group structure is given. This model captures the main characteristic features of a group structure and functionality of its components. Second, Multiple Object Invocation is presented. It is based on a simplified group structure model and may be used as a basis for creating group services according to the general model. Multiple Object Invocation have been implemented in Emerald. Emerald is an object-oriented language and distributed operating system and as such may be viewed as an object-based environment. A description of implementation of MOI in Emerald concludes the paper and shows how a general model and the idea of MOI may be applied.

2 A general model of a group

In this section a general model of a group structure is proposed. This model provides a scheme which may be used for expressing interactions involving an arbitrary number of participants. Its application orders entities according to their functionality, exhibits logical links between them and allows hierarchical structuring of groups of objects. In this way, the model may simplify specification and add to more appropriate design of complex systems.

Throughout this section the term object denotes any entity taking part in a communication process, be it a language-level object, a process, a module, or a processor.

2.1 Groups and group actions

A group is a set of objects performing collectively some actions. The group's actions may be performed as a response to clients' requests, other external events or may be initiated by the group itself. Group actions may involve communication among cooperating objects - also group communication.

Group communication is a single act of information exchange involving any number (possibly more than two) of objects. It is based on passing messages to or receiving them from a number of objects (many-to-many, one-to-many, many-to-one) as opposed to traditional communication (one-to-one) in which only two objects interact (see Figure 1).

Just like there are several types of simple communication (synchronous and asynchronous message passing, rendezvous, remote procedure call), some typical schemes of group communication may be found. The simplest instance of group communication is passing a message to a number of objects (multicast or broadcast). The second example is multiple process rendezvous which is a synchronized exchange of messages among any number of processes. The third possibility is concurrent invocation of a number of procedures in which a number of new threads of control is created to

Group Communication in an Object-Based Environment

Przemysław Pardyak *†
Department of Computer Science - University of Copenhagen
Universitetsparken 1, DK–2100 Copenhagen Ø, Denmark

Abstract

This paper presents a proposal for incorporating a group communication paradigm into an object-based environment. Group communication is an interaction involving an arbitrary number of participants. Its application simplifies specification and implementation of complex systems. Object-oriented programming can significantly profit from the addition of a group mechanism. Our proposed mechanism, Multiple Object Invocation, provides group communication among objects. The paper first presents a model of general structure of group services and the use of group communication in this model. Multiple Object Invocation is presented and its characteristics and application in the general model is discussed. Implementation of Multiple Object Invocation in the Emerald object-oriented language and distributed operating system is presented.

1 Introduction

Many computations require exchange of information among entities taking part in them. In general, such communication take one of two forms depending on the number of participants: a trivial case, in which only two entities participate, or a more complex one, when an arbitrary number of participants is involved. The first form is essentially a normal point-to-point communication such as message passing or a remote procedure call. It is used extensively and is well understood, both as a means of communication in distributed systems and in terms of language constructs; therefore we call this form traditional. The second form, which is referred to as group communication, is more powerful than the traditional form and has recently received much attention.

*On leave from University of Mining and Metallurgy, Cracow, Poland.

†This work was partially supported by the Danish Natural Science Foundation under grant no. SNF 11-7768.

This paper addresses issues related to communication among groups of objects. It presents a general model of a group structure and proposes Multiple Object Invocation (MOI) as a realization of group communication in an object-based environment.

1.1 Object-based environment

An object-based environment may be described as a system in which all first-class entities are objects. Such an environment allows abstraction from other notions like process, data, file, etc., and treat them uniformly. The objects used in such an environment do not have to be language-level objects but if a system is to be object-based, it has to consist exclusively of encapsulated modules accessible via a strictly defined interface by means of invocation of methods given in the interface.

1.2 Groups in operating systems

Object-based operating systems are good examples of object-based environments. They can be thought of as a collection of services offered to application programs and implemented as objects. The system according to this metaphor is a set of communicating objects.

All tasks performed by the system may be split into two categories. The first category includes individual tasks, i.e., those which are fulfilled by one server or which may be decomposed into a number of subtasks done by some servers on behalf of other. The second category includes collective tasks, i.e., those that are carried out by a number of objects which cooperate to perform their common task using communication for information exchange and achieving synchronization. The examples of such group services may be load balancing, distributed name service, or any replicated server.

The traditional communication strictly corresponds to individual tasks and serves well for achieving them.