Multicast RPC in Extended-C++

Michael Olsen, Robert Seliger
Intelligent Networked Computing Laboratory
HP Laboratories Bristol
HPL-93-84
September, 1993

remote procedure call, multicast, distributed systems, object-oriented, C++

Multicast RPC enables one object to make a single remote procedure call which is in fact transparently executed in many remote objects. This report describes a design and a prototype implementation of a software architecture which is used for supporting multicast RPC in the Extended-C++ programming language [Seliger 90]. It also describes how this architecture is enhanced to ensure atomicity of a multicast RPC, i.e. either all of a collection of functional objects execute a called procedure or none of them do. Extended-C++ is a production programming language which is used for developing CareVue, a distributed clinical information system product.

1 Introduction

The task of building distributed applications is facilitated by remote procedure calls (RPC). RPC is a mechanism which transfers control and data across a communication network, to enable an object in one address space to call the procedures of objects in other address spaces using syntax and having semantics similar to a local procedure call [Birrell 84].

Multicast RPC takes this a step further. It enables one object to transparently send a single logical message which is in fact received by many remote objects. This capability, particularly when it is reliably implemented and supported by a useful set of correctness guarantees, can be the basis for:

- providing a richer substrate for the creation of event-based distributed applications;
- simplifying the implementation of distributed and replicated software components;
- providing a mechanism which increases the amount of decoupling and independence between distributed objects to simplify designs and increase system robustness.

In many distributed systems[1], system or application services often make use of multicast communication which is emulated by explicit implementations built on top of other communication primitives. The intent of providing platform support for multicast RPC is to offer a more unified and robust service which provides a richer set of capabilities and guarantees than can be implemented by the typical developer of a service or application for a distributed system.

Extended C++ is an object-oriented programming language which extends C++ with support for RPC, concurrency, exception handling, and garbage collection [Seliger 90]. It was developed to facilitate the implementation of CareVue, a distributed clinical information system product which is used in hospital critical care environments.

This report describes (i) how multicast RPC has been incorporated in Extended-C++, (ii) how it is engineered in a software architecture, (iii) how this architecture can be extended to ensure atomicity of multicast RPC, and (iv) a prototype implementation of multicast RPC in Extended-C++.

Multicast RPC is supported in Extended-C++ by introducing the abstraction of a group (of objects) [Birman 91]. The group abstraction is based on ideas from ANSA [Olsen 91] and it is engineered using ideas from group communication in Amoeba [Kaashoek 91].

Within the research community, the group abstraction is now well understood, and general support mechanisms have been demonstrated in prototypes. Extended-C++ is used for production programming, so its support for a group abstraction is pragmatic in terms of favouring simplicity and efficiency rather than generality.

---

1. By *system* is meant a collection of objects which share a common symbolic name-space and which are capable of communicating with each other.

The main results of this report are:

- multicast RPC can be supported by introducing very few language level extensions;
- the engineering of groups can be substantially simplified by assuming a reliable service for group management;
- the number of constituent calls of a multicast can be minimized by organizing the engineering of groups hierarchically;
- hierarchical organization of groups enable the group abstraction to scale up well;
- the implementation of multicast RPC can be substantially simplified by building it from an existing RPC implementation;
- the performance of the multicast RPC prototype implementation is extremely good despite being built using existing RPC, and despite providing a strong ordering guarantee on the order of call execution in multicast callees.

2 Aims and requirements

The aim is to provide a simple and efficient way for objects to call a procedure on a group of objects via multicast RPC. Without simplicity and efficiency, application programmers are less likely to use multicast RPC, and more likely to build their own multicast facilities within applications.

2.1 Simplicity

To ensure simplicity, the introduction of multicast RPC must not compromise the existing object model supported in Extended-C++. Multicast RPC must therefore

- support a fine-grain object model, — objects are C++ objects, not whole address spaces;
- provide a statically typed interface to groups of objects, similar to the interface to a single object which is provided by Extended-C++ RPC.

To enable ease of use, it must also provide easy understandable semantics, therefore it must

- guarantee total ordering across all calls received by the objects in a group and any change to the number of objects in a group.

Figure 1 illustrates how we intend to provide a simple and uniform multicast RPC facility in Extended-C++.

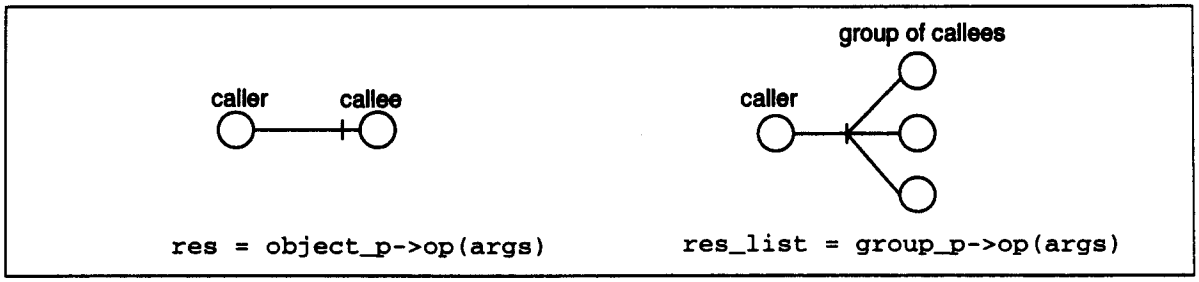

Figure 1. RPC and multicast RPC invocations

A remote procedure call is programmed by

```
res = object_p->op(args);
```

where `object_p` is a handle to the remote object on which `op` is called with some arguments. The caller is blocked while the call is carried out, and if the call is successful then it may return a result which is assigned to `res`.

The intention is to be able to program a multicast RPC in Extended-C++ in a way which is syntactically very similar to the programming of a remote procedure call, that is

```
res_list = group_p->op(args);
```

where `group_p` is a handle to a collection of remote objects on which `op` is called with the provided arguments. The caller is blocked while the call is carried out, and if the call is successful then the results which may be produced in each remote object is collected in a list which is assigned to `res_list`. The ordering of the execution of the called procedure is the same in all the objects which are called by multicast RPC.

2.2 Application requirements

Many CareVue applications are event-based and require that replicated objects maintain synchronized state. Multicast RPC in Extended-C++ should therefore

- support communication capabilities which facilitate the development of event-based applications (i.e., applications which communicate asynchronously but which still require reliable message delivery);
- support communication capabilities which facilitate the development of replicated objects which maintain synchronized state.

Replicated objects can be used for increasing the fault-tolerance of a distributed system in which components can fail independently. The aim is thus to be able to use multicast RPC for implementing atomic delivery of a call to all or none of a group of objects.

3 Object groups in Extended-C++

Support for multicast RPC in Extended-C++ is provided in terms of the *group* abstraction [Kaashoek 91][Birman 91][Olsen 91]. A group is a named, typed, interface instance which represents a collection of objects. Different groups can have different types, and groups of the same type can have different names. The possible types of groups is statically defined at compile time, but a particular group with a particular name is formed at run-time by instantiating it from a group type. Objects which wish to join a particular group must call a procedure *join* on the group type and specify the name of the group they will be joining. A static type check ensures that the type of a joining object, i.e. the collection of signatures of its operations, is a subtype of the group, i.e. the collection of signatures of operations that constitutes the group type. Objects which wish to leave a particular group must call a *leave* procedure on the group type and specify the name of the group that they belong to.

In traditional RPC, communication is point-to-point between a client (caller) and server (callee) object. Although the client does not know the location of the server object, it still must identify (via a handle) the server object of interest. In contrast, group communication is between a client and a named group interface instance which represents an anonymous collection of objects which may be local or remote. The named group interface instance is (logically) responsible for the dissemination of received calls to the objects which belongs to the group. It thus provides an indirection point which decouples knowledge about how to invoke a collection of objects from knowledge about which objects belong to the collection. The difference between point-to-point RPC and group communication is shown in Figure 2.

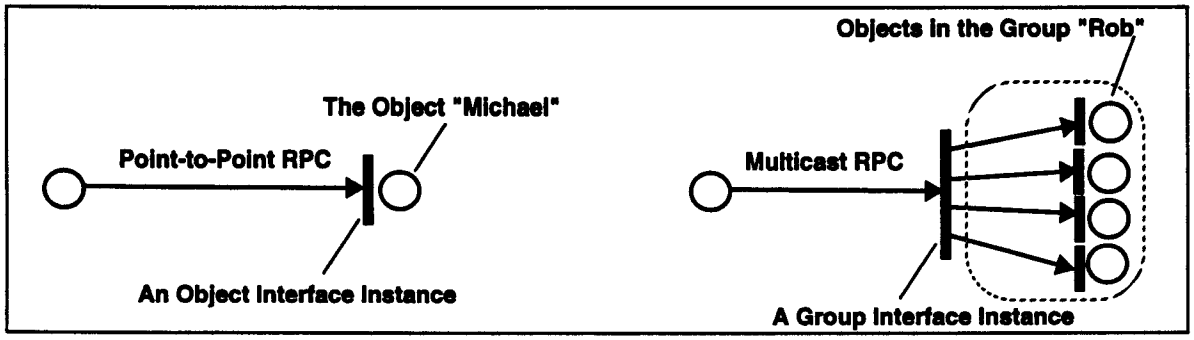

Figure 2. Point-to-point RPC versus group communication

Names of object instances and group instances are used to enable objects to obtain handles that they can use for calling the procedures of objects and groups. In the case of groups, a group name is also a means for an object to identify which group it wishes to join or leave.

The effect of calling a procedure on a group, i.e. multicast RPC, can be simulated by making a RPC call to a proxy object which in turn makes a sequence of RPC calls on the objects in the group. Multicast RPC ensures that the order in which the objects in a group dispatch and execute calls is the same in each object in the group, i.e. a total ordering. To simulate this with the proxy, the proxy must complete a multicast before it can initiate another one, i.e. it must act as a sequencer (see Section 4.1).

Extended-C++ RPC supports the ability to block a caller until the callee has dispatched, executed and possibly produced a result which is returned to the caller (synchronous RPC), and the ability to block a caller until a call has been delivered but not dispatched and executed by the callee (asynchronous RPC). Asynchronous RPC cannot return a result to the caller.

In a similar way multicast RPC can be either synchronous or asynchronous. Synchronous multicast RPC blocks the caller until all objects in a group have dispatched, executed and possibly produced a result all of which are returned to the caller. Asynchronous multicast RPC blocks the caller until all objects in a group have received, but not dispatched nor executed a call. Asynchronous multicast cannot return a result to the caller.

4 Properties of multicast RPC

The properties of multicast RPC can best be illustrated when there are more than one object which calls a group. Figure 3 contrasts a handcrafted solution to using multicast RPC, in the situation when there are two callers.

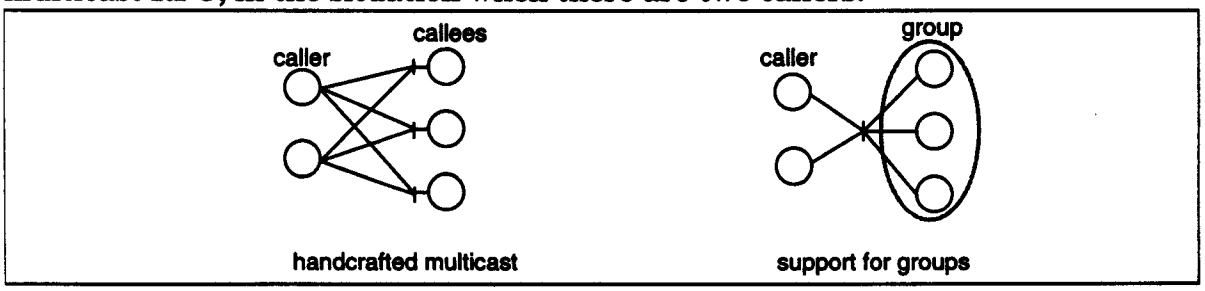

Figure 3. Handcrafted multicast contrasted to support for groups

With handcrafted multicast, a caller must obtain a handle to each object it wants to call; it must spawn threads if these objects should be called concurrently, and it must deal with each RPC failure that occurs. Multiple callers to the same set of objects need to have their calls synchronized, if there is a requirements on the order in which calls are received in each callee (for example, if each callee should execute all calls in the same order, as the other callees). Whenever the set of callees need to change, all callers need to modify the set of handles that they possess to callees, and they may need to do so in a synchronized way.

With support for groups, each caller possesses a single handle, the group handle, to a group of objects. When a caller calls the group, threads are transparently spawned to call each group member concurrently. Each caller need only deal with a multicast RPC failure. All calls made via the group handle are totally ordered, so that the sequence of calls that a group member executes is the same as that of all other group members. In addition, any membership changes to the group are totally ordered with other calls to the group. The handle to a group does not depend on which objects belong to a group. This means that group handles require no modification when the group undergoes membership changes.

The independence between a group handle and the membership of a group is an important property, because it increases the amount of decoupling and independence between distributed objects thus enabling simpler designs and increased system robustness.

4.1 Ordering

The total order guarantee that group support provides is illustrated in Figure 4. With no order guarantee, and the two callers being independent, it is possible that callee 1 has b called before a, whereas callee 2 has a called before b, and callee 3 has b called before a. Any permutation of a and b calls are possible in each of the callees.

With the total order guarantee provided by multicast RPC, and the two callers being independent, callee 1, 2 and 3 will all have b called before a (as in the figure), or will all have a called before b, depending on whether the multicast RPC protocol selects a or b to be executed first.

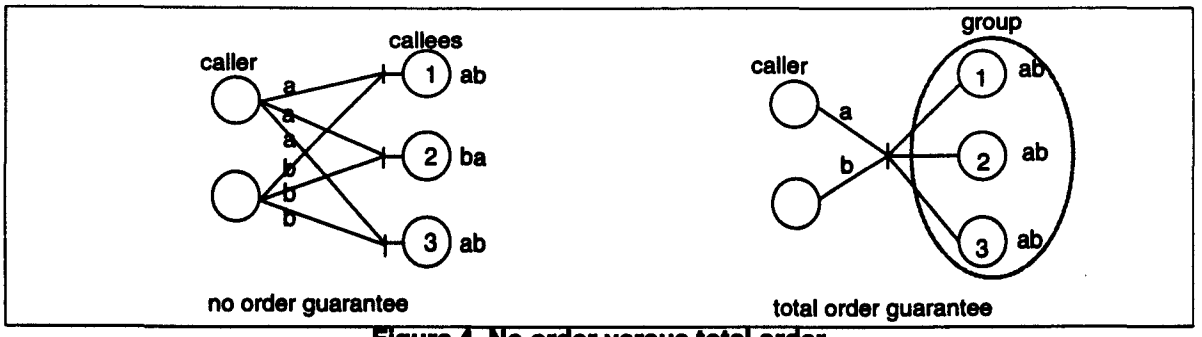

Figure 4. No order versus total order

5 Group features for multicast RPC in Extended-C++

This section describes the group features that have been designed for the prototype implementation of multicast RPC in CareVue.

The keyword **group** is introduced in Extended-C++ to distinguish the specification of a class, which implicitly defines an object interface by the operations in the class, from the specification of a group interface type. The keyword is to be used for declaring group interfaces and for instantiating groups from them.

The instantiation of a group results in the construction of a group with no members, and a handle to the group is returned. A group instance can be destroyed by invoking a destroy operation on the group, but the effect is delayed until the group has no members.

A group handle can be passed around similarly to the passing of handles to remote object. This enables group handles to be inserted in name servers for use by unknown third parties, and to be passed between objects.

5.1 The features

A group (interface) with a single operation `op1` can be declared as follows:

```
group ABC
{
  public:
    void op1(String);
};
```

A group can, as opposed to a class, only have operations as members. It would not make sense to allow a group instance to have state (what object should encapsulate it?).

A group handle can be instantiated, and assigned to a group pointer as follows:

```
group ABC *abc_p = ABC::form("Olsen");
```

The `form` operation is a standard operation which is inherited in all groups. Its execution results in the instantiation of a group which is identified by the argument string, in this case a group instance called "Olsen". However, if there already exists a group instance which is identified by the given argument string, then this instance is returned as the result of invoking `form`.

An object can be made a member of a group, if it is instantiated from a class which is a subclass of a group. E.g.

```
class XYZ : public ABC
{
  public:
    XYZ();
    void op2(int);
}
XYZ *xyz_p = new XYZ;
```

The object pointed to by xyz_p can now be made a member of the group identified by "Olsen" by calling the standard group operation join as follows:

```
abc_p->join(xyz_p);
```

The object pointed to by xyz_p can subsequently leave the group by calling the standard group operation leave as follows:

```
abc_p->leave(xyz_p);
```

A group handle can be destroyed by calling the standard group operation destroy as follows:

```
abc_p->destroy();
```

For performance reasons, the group operation form should only be used when a group handle will be used for joining objects as members of the group. This is because the execution of form will cause all multicast messages, that are intended for the group, to be sent to the address space in which form is executed (see Section 6.4.1).

If a group handle will not be used for calling join or leave operations on a group, then it should instead be instantiated via the standard group operation locate. Using locate, a group handle can be instantiated, and assigned to a group pointer as follows:

```
group ABC *abc_p = ABC::locate("Olsen");
```

The locate operation is inherited in all groups. It differs from form, in that it does not result in the instantiation of a group if there is no group instance which is identified by "Olsen", and it does not cause multicast RPC messages intended for the group to be sent to the hosting address space. If there is no group instance identified by "Olsen", then locate returns a null pointer[1].

Note, how the use of locate for obtaining a group handle provides a type-safe way of binding group handles to pointers. This is in contrast to the cast that needs to be made, when handles are obtained via primitive name servers. Also note, that it is possible to obtain a group handle via form, without using it for joining members. By not joining objects as members in the address space where the execution of form resulted in the construction of a new group, it is possible to achieve a performance gain, because there are no members that need scheduling in the address space (see Section 6.2).

A group instance can be called via a group handle as follows:

```
abc_p->op1("Hello World");
```

This call is a multicast RPC which causes all members that belong to "Olsen" to have op1 called with "Hello World" as argument.

Note that

```
abc_p->op2(42);
```

1. It may be useful to provide the locate operation with an optional time out or wait-forever parameter.

will be caught as an illegal operation during static type checking, because op2 is not in the group interface.

It is possible to define asynchronous operations[1] in a group interface, e.g.

```
group ABC
{
  public:
    op1(String) threadable;
};
```

This has the effect that an object which calls the group, is blocked only until each member has received the call, but not while the members are dispatching and executing it.

A group operation can be defined to return a result. This result needs to be a list of the results returned by each object in the group. Because the operations that will be called on the objects in a group must be subtypes of the operations in the group interface, the operations in objects must return a result as a list with one element. For example, if a group interface type is specified as follows:

```
group ABC
{
  public:
    Result(int) op1(String);
};
```

where `Result(int)` is a list parameterized with the type of each element, in this case `int`, which is equivalent to the type

```
struct {
  int: length,
  int: *data} Result;
```

then a call to `op1` on a group instance that has 3 members returns a list of length 3. This list is produced by gathering together the lists of length 1 that each member returns.

The above is only one out of many possible ways of enabling group operations to return results to the caller. A discussion on group results can be found in [Cooper 90]. Of course, the programmer can always construct a group collector for collecting replies from the members of a group as part of the caller program, and then pass the collector's handle as an additional argument when calling group operations.

Each group member can then use this handle to pass a result back to the caller's address space, where the caller can obtain a result that the collector produces from the individual results.

Of the group features described above, `leave`, `destroy`, and group results have not been incorporated into the prototype implementation, see Section 9.

An example application, which runs in the prototype implementation, is shown in Figure 5, where the execution of program A will result in the members of the group "G1" being called once, and where each execution of the B program will result in a member being added to "G1".

1. Extended-C++ provides the key word `threadable` which when used as a predicate on an operation declaration, declares that the operation will be executed by threads which are independent of the calling threads.

program A - caller

```
group ABC
{
  public:
    void op1(String);
};

main()
{
  group ABC *abc_p = ABC::locate("G1");
  abc_p->op1("Hello World");
}
```

program B - callee

```
group ABC
{
  public:
    void op1(String);
};

class XYZ : public ABC
{
  public:
    XYZ();
};

main()
{
  group ABC *abc_p = ABC::form("G1");
  XYZ *xyz_p = new XYZ;
  abc_p->join(xyz_p);
}
```

Figure 5. Example programs for a multicast RPC caller and callee

5.2 Transparency

The use of groups is not transparent in Extended-C++ for the obvious reason, that the programmer must use the `group` keyword (when specifying group interfaces and when instantiating groups). This is in line with the non-transparency in Extended-C++ between handles used for local procedure calls and handles used for RPC. The reason for making that distinction is to make the programmer aware of the differences in failure modes and latency. Multicast RPC is differentiated from RPC for the same reasons. This deviates from the approach taken in ANSA and ANSAware [ANSA 89], where there is no computational notion of local calls.

In Extended-C++, a call itself does not reveal whether it will result in a RPC or a multicast RPC being carried out. Both RPC and multicast RPC provides location transparency (it is not known where the remote object(s) are located). With multicast RPC it is also transparent which objects belong to a group.

6 The architecture for engineering groups in Extended-C++

The software architecture for the engineering of groups comprises (i) a set of hierarchically organized components, (ii) multicast logic for ensuring reliably delivery and total ordering of calls executed by the objects in a group, and (iii) management logic for maintaining the membership information of a group. The hierarchical organization of components serves to reduce the message traffic of the multicast logic, and to reduce the need for membership information in each component.

The design of the software architecture is based on three assumptions each of which simplifies the design task significantly.

Robust Logically Centralized Service

It is assumed that there is an very robust, logically centralized, service which can be used in lieu of (complex) truly distributed, algorithms. The idea is to define a reasonably small set of responsibilities for this service which assumes that the service rarely fails and that the service will not become a performance bottleneck as its responsibilities are exercised. In doing this, the intent is to simpify the

overall problem of designing a multicast solution by making a small set of simplifying assumptions which can in turn be supported by handcrafted, special purpose, solutions.

Basic RPC Capabilities

It is assumed that basic RPC capabilities include:

- a highly reliable service with minimal likelihood of premature time-out due to network or processor congestion;
- at-most-once semantics;
- robust clean-up of client and/or server state if a client or server fails in the middle of performing an RPC.

Malleable White Box RPC Protocol

It is assumed that the underlying RPC protocol can be adapted and enhanced to offer capabilities which are tailored to support multicast communication. Treating the RPC mechanism as a malleable white box can also offer opportunities for making end-to-end design trade-offs which result in simplifications.

6.1 Components of the software architecture

A group is engineered by the following components:

Group Member - an object which belongs to one or more groups; there can be many Group Members in a system;

Group Representative - represents a collection of zero or more objects in a particular address space which are members of a particular group; there is one Group Representative per group per address space;

Group Leader - a distinguished Group Representative which is responsible for managing Group Representatives and for sequencing all calls to the group's operations; there is one Group Leader per group per system;

Group Manager - a logical object which contains the names of and lists of Group Representatives for all of the groups in a system; there is one Group Manager per system.

The caller of a group, a Group Client, of which there can be many for each group, does not belong to the components that make up the software architecture for groups.

A Group Representative and the Group Members that it represents reside in the same address space. The Group Manager logically resides in a address space which is assumed to be extremely robust, but it may be physically realized by a collection of distributed objects that reside in different address spaces. A Group Leader and a Group Client may reside in the same or different address space; either can reside in the same address space as a Group Representative and some set of Group Members.

A typical distribution between these objects is shown in Figure 6.

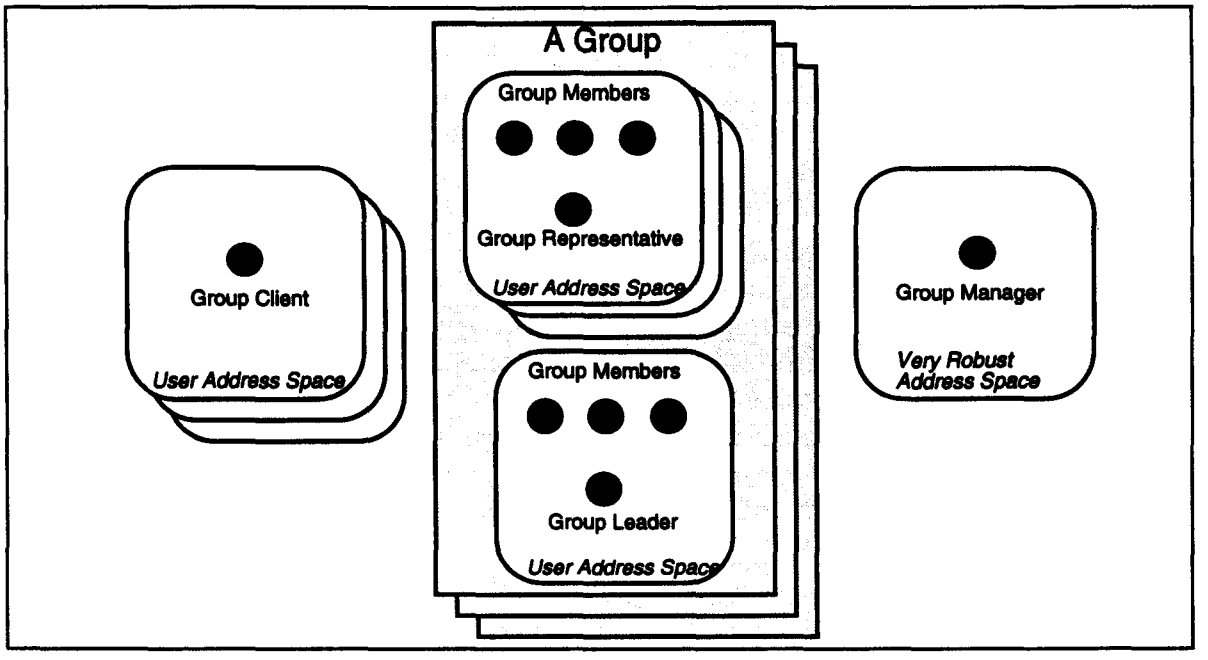

Figure 6. Typical Distribution of architectural components

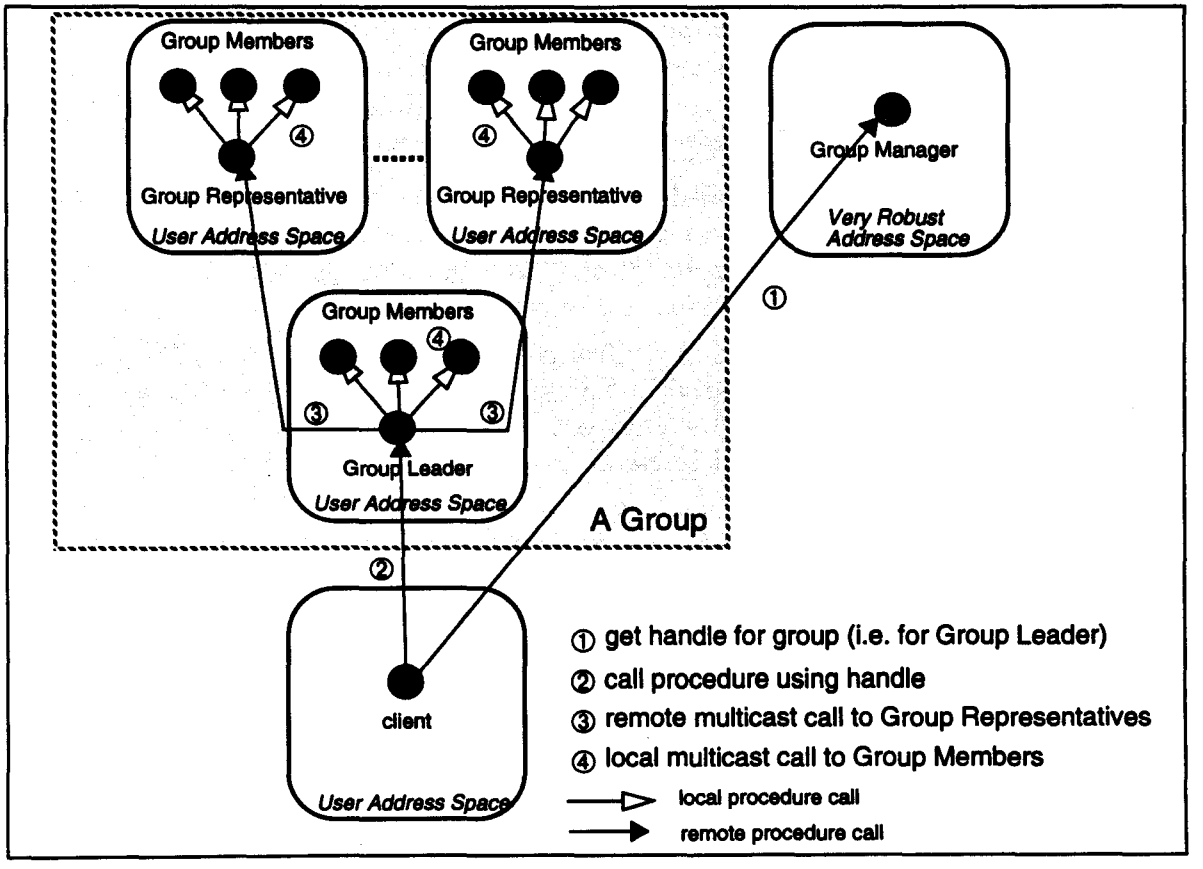

Figure 7. Group Invocation Logic

6.2 Multicast logic

The term *invocation* is used in the remainder of the report to denote the act of calling a procedure on a group. An invocation is implemented by an exchange of some set of inter-object messages which performs the distribution and coordination necessary to deliver a call to each of a group's members. An invocation originates when a client object executes a procedure call on a group. Once an invocation is complete, the called procedures can be *dispatched* and *executed*.

Assuming that a group has formed and that there is a leader, a client obtains a group handle by calling `locate` with an argument that denotes a group. The result of this call is a handle to the leader of the group, which is obtained from the Group Manager. Using this handle for an invocation, the client will in fact, but unknowingly, call a procedure on the leader, which in turn distributes this call to each of the group's representatives. Each representative then distributes the call to the members which are co-resident in the representative's address space. This hierarchically organization of the group components by the multicast logic is diagrammed in Figure 7.

There are several key points about this hierarchical organization of the architectural components

1. By having all of a group's invocations pass through the group's leader, it is possible to enforce a total ordering of all the invocations that are received by the groups members. To accomplish this, the leader simply blocks the next invocation until the current invocation has been completed. Once the invocation has been completed, the dispatching and execution of the function which was called can be synchronous or asynchronous depending upon the required semantics and desired parallelism between the group members.
2. A group with M members and R representatives, where R <= M, requires[1] at most 1 + R remote messages per invocation. This is only one more message than the theoretical minimum of one remote message per address space (R messages), and can be significantly less than M messages.
3. The hierarchical organization of Group Members and Group Representatives has the effect, that membership knowledge is local to the address space in which members reside. The Group Leader only knows about the group's representatives, it does not know of Group Members other than those that it represents itself.
4. Multicasts to the members of a group and membership changes are coordinated by the Group Leader and Group Representatives. The leader can serialize multicasts with changes to the set of representatives, and representatives can serialize multicasts with changes to the set of members they are responsible for.
5. At-most-once delivery of an invocation is achieved by relying on at-most-once RPC semantics, and by restricting an object from being able to join a particular group if it is already a member.

---

1. Assuming multicast is implemented using existing RPC. If true multicast is available at the transport protocol level, it can be used here.

6. The packaging of the code which actually implements a group and the computing resources consumed by the group can be readily separated from the code and resources needed for clients. This allows the cost of using a group to be kept to a minimum for group clients, and it allows changes to group implementations to be made without impacting client programs.

In Section 8 it is shown, that although the Group Leader is a singular focus for group activity, it need not become a single point of failure.

6.3 Result collection logic

If a Group Member returns a result, this can be passed to the client by first passing the result to the Group Representative which passes it to the Group Leader which passes it to the client. This scheme reverses the invocation logic.

It is up to an implementation to work out an appropriate result collection scheme, of which there may be many options within a single implementation.

6.4 Management logic

The management logic is responsible for managing the components that engineer a group.

6.4.1 Group Representative

A Group Representative manages the Group Members that reside in the same address space as itself. The `join` and `leave` operations which are called by Extended-C++ objects on a group are executed by the representative which resides in the same address space as the calling object.

A Group Representative is instantiated when an Extended-C++ object calls the `form` operation on a group type for which there is no representative in the address space of the calling object. When a representative is being instantiated it requests the Group Manager for a handle to the Group Leader of the group. The representative uses the handle to request the leader to be joined to the set of representatives for the group.

6.4.2 Group Leader

A Group Leader has two roles (i) it is a Group Representative for the Group Members which reside in its address space, and (ii) it manages the other Group Representatives. The leader serializes Group Representative membership changes with group invocations. It is thus ensured that these membership changes are atomic (i.e., they occur in-between invocations), and that they are part of the same total ordering that governs group invocations.

6.4.3 Group Manager

The Group Manager manages the Group Representatives for each group, and it registers which Group Representative is the Group Leader of each group. When a representative is instantiated, it requests the Group Manager for a handle to the Group Leader, and if there are no leader for the group, the Group Manager appoints the representative as leader. It is the Group Manager from which a client obtains a handle to the current leader of a group.

The Group manager must have been instantiated before any groups can be instantiated. When a Group Manager is instantiated, it publishes a handle to itself in a name server.

6.4.4 Group Leader termination

Group Representatives and Group Members can terminate without affecting the service of a group which is provided by remaining Group Members, because their request to leave (and join) a group are totally ordered with invocations on the group. A Group Leader can terminate by simply choosing to no longer process new invocations. Clients which attempt new invocations will receive an exception (e.g., due to an RPC failure), and can then contact the Group Manager for a handle for the new leader (as detailed in Section 7.1.1).

7 Replication

When a replicated service is provided in terms of a group in which each Group Member is a replica which implements the service, it is necessary to ensure that the state of the Group Members are synchronized. Extended-C++ supports the provision of encoder and decoder functions which are operations in a class that perform marshalling and unmarshalling of the state of an instance of the class. These operations must be defined by the application programmer so that the group infrastructure can copy the state of one of the Group Members to an object which is joining the group.

The synchronization of the state of objects that join a group with the state of an existing Group Member is carried out by the Group Representative which receives the join request. When a Group Representative receives a join request it calls a state encoder operation on one of the Group Members that it represents and installs the state in the joining member by calling its decode operation with the encode state as argument. If there are no Group Members currently being represented by a Group Representative, then the Group Representative requests the Group Leader to return the encoded state of a Group Member. The Group Leader replies to this request by obtaining the encoded state from one of the Group Members that it or one of the other Group Representative represents. If there are no Group Members present, then the object that joins a group defines the state that Group Members must maintain.

8 Reliability

This section describes how reliability is addressed by the group architecture. The basic reliability objective for the architecture is to ensure that the scope of address space failures can be constrained, so that, in general functional objects can continue to make progress. Specifically:

- A client's failure is atomic - either the invocation it has made is executed by all or none of the group's members (i.e. the group's members discard the invocation). Invocations made by the client, which have been queued by a Group Leader (as part of the serialization of invocations) can be discarded by the Group Leader.

- The failure of a Group Leader always results in an attempt to chose a new leader (which may not succeed if there are no other Group Representative capable of becoming the leader). Failure of the leader in the midst of an invocation is not necessarily atomic: some members may receive the invocation, others might not, and the client will see an exception instead of a reply from the group. (Section 9 describes support for stronger guarantees.)
- The failure of a Group Representative does not affect members that it does not represent, other representatives, or the Group Leader, but it may cause the invocation to fail (client sees an exception) because it may not be possible for the group to provide its delivery guarantee.
- Except when a software defect is to blame (e.g., the premature destruction of a Group Representative or Group Member), a Group Representative and its Group Members always fail together (because they reside in the same address space).
- The failure of one Group Member does not affect other Group Members, Group Representatives, or the Group Leader, but it may cause the invocation to fail (client sees an exception) because it may not be impossible for the group to provide its delivery guarantee (e.g. at least N Group Members receive an invocation)..
- The Group Manager is assumed to fail rarely; when it does, a system restart may be required to recover the Group Manager.

The remainder of Section 8 describes the various failure scenarios and protocols for dealing with them.

8.1 Group Leader fails

A Group Leader can fail during an invocation, or between invocations. In both cases a new leader must be chosen, but in the former case, the client and/or the Group Representatives must deal with a potentially interrupted RPC.

The failure of a leader can be detected in three ways: (i) a client is unable to communicate with the leader, (ii) a group representative notices, while receiving a invocation or transmitting a reply, that the leader has failed, or (iii) the Group Manager (which can detect the existence of address spaces in the system) notices that the address space that contained the leader no longer exists.

If a Group Representative notices that the leader has failed, it sends a message to the Group Manager requesting that a new leader be chosen. If the Group Manager notices the failure, then it decides to chose a new leader. The simplest way for the Group Manager to chose a new leader is to simply consult its membership list for the group and pick one of the remaining Group Representatives which is capable of being the leader. This object is then informed that it is now the leader.

A client may also notice that a leader has failed, in which case it contacts the Group Manager to request a new handle. If the client's current handle matches the handle for the object that the Group Manager currently regard as the leader, then the Group Manager must choose a new leader. If the client's handle does not match, then the Group Manager simply returns the correct handle to the client.

This simple reformation strategy obviates the need for objects other than the Group Manager and the newly chosen leader to be involved in the reformation. The other group representatives can just wait passively until a new leader is chosen.

The Group Manager may receive many requests to elect a new leader when the Group Leader has failed. These requests must provide enough information to enable the Group Manager to detect when a request is already accommodated by a previous request.

The maintenance by the Group Manager of Group Representatives membership lists which are consistent with those maintained by the various Group Leaders is accomplished by using a log-then-update algorithm similar to that which is employed in database systems. The idea is for a Group Leader to first inform the Group Manager of any membership changes, after which the Group Leader changes its own membership list, and then informs the Group Representative which requested the change that the change has been granted.

If the Group Leader fails, then the Group Representative which requested the membership change can try again with the new leader. The new leader will either already be aware of the membership change based upon the membership list it received from the Group Manager (because the old leader failed after telling the Group Manager about the change), or the new leader will simply perform the membership change.

8.2 Group Representative Fails

A Group Representative can fail during an invocation, or between invocations, but in the latter case the failure may not be detected (by the leader) until the next invocation if the invocation returns no reply. When the leader detects the failure, it removes the representative from the membership list. It then ceases to include the failed representative in the current and subsequent multicasts. The failure may prevent the leader from providing the group's delivery guarantee, in which case the client will receive an exception.

8.3 Client fails

The only client failures which matter to a group are the ones that occur *during* an invocation. If the client fails before sending the entire invocation, then the leader simply discards the invocation in progress. If the leader has started to multicast the invocation to the representatives, then they also discard the invocation. Until the entire invocation has been transmitted, it is not possible for a representative to have dispatched the remotely invoked function on the members it represents, so the only clean-up necessary is to destroy any (partially) unmarshalled RPC arguments[1].

1. A representative would not need to clean-up at all if the leader did not start a multicast until it had received an entire invocation, but this would require the leader to be able to buffer the entire invocation and incur the memory overhead for maintaining a potentially large amount of transient data.

If the client fails after the invocation has been completed (but before the reply has been received), then the orphan invocations are allowed to run to completion, and the group's reply is then discarded by the leader.

9 Atomic multicast RPC

Atomic invocation groups are groups which have the additional guarantee that if a group's client or leader fails in the midst of an invocation, then either all of the group's extant members receive the invocation, or none of them do. Note that this form of atomic invocation does not deal with failures of group members during an invocation. In other words, the failure of a group member during an invocation will not affect the invocation of the remaining group members. Atomic invocation groups are particularly useful for implementing replicated objects where the idea is to be able to tolerate the failure of a replica.

Atomic groups are an extension to the basic groups described so far. Atomic invocation is provided by maintaining enough information for an invocation such that, in the advent of a failure of the group's leader in the midst of multicasting a message, the members which have not received the entire message can obtain it from another source, specifically, the Group Manager.

It turns out that the entire message does not need to be stored by the Group Manager, and that the information that is stored needs to only describe aspects of the most recent invocation on the group. The latter observation is based upon the fact that Group Leaders already serialize invocations, so that there is at most one invocation per group in progress at a time. The former observation is based upon knowledge of how the underlying RPC protocol works, wherein the last packet must be received before the function which was (remotely) called will be dispatched.

9.1 Implementing Atomicity

Atomic invocation groups are implemented by having the Group Leader send the *last* packet of the current invocation to the Group Manager *before* it sends this packet to the various Group Representatives. The Group Manager maintains this packet until the next *last* packet for the next invocation for the group is received. There are several failure scenarios to consider:

1. The Group Leader fails before sending the *last* packet to the Group Manager. None of the Group Representatives have received the *last* packet, so none of the members can proceed with dispatching and executing the called procedure. The RPC protocol allows receivers to detect the failure of senders (as well as vice-versa), so each representative unilaterally discards the invocation. The invocation has, in effect, never occurred.
2. The Group Leader fails after sending the *last* packet to the Group Manager but before any Group Representative has received the last packet.
   Each Group Representative contacts the Group Manager to obtain the *last* packet and proceeds with the invocation. Prior to choosing a new Group Leader, the Group Manager reliably sends the currently buffered last packet

to all of the group's representatives so that they have a chance to complete the current invocation prior to a new invocation via the new leader.

3. The Group Leader fails after sending at least the *last* packet to at least one Group Representative.
   Same as #2, but only the Group Representatives which did not receive the last packet contact the Group Manager to obtain it; all invocations still run to completion.
4. A Group Representative fails during an invocation.
   The RPC detects the failure of the Group Representative and the leader ceases to include the representative in its multicast; the leader is eventually informed by RPC run-time that the representative has failed, so it removes the representative from the membership list.

10 Prototype implementation of multicast RPC

The purpose of developing a prototype implementation of CareVue platform support for multicast RPC are multi-fold:

- to justify the most basic functionality of the multicast RPC architecture;
- to obtain a demonstration and experimentation vehicle that effectively shows how multicast RPC support can simplify many types of applications;
- to provide performance figures that show that the performance of applications which are built with multicast RPC support does not have an overhead which renders platform support for multicast RPC impractical;
- to provide an implementation which can serve as a basis for introducing platform support for multicast RPC support in forthcoming releases of the CareVue 9000 product.

The prototype implementation comprises modification to the Extended-C++ preprocessor to support the language extensions for groups, a multicast extension to the existing RPC library, and a group management library.

10.1 Simplifications

To reduce the prototyping task without reducing its proof-of-concept value, the following simplifications apply to the prototype:

- group operations cannot return results;
- once joined a group, a member cannot leave the group;
- the state of objects that join a group is not synchronized with the state of existing group members;
- once instantiated, a group cannot be destroyed;
- no handling of failures.

10.2 Multicast logic

The current Extended-C++ RPC implementation marshals arguments into one or more RPC packets. Upon receipt, the arguments are unmarshalled from the packets and the packets are discarded. For the Group Leader to propagate an

invocation to the Group Representatives efficiently, it needs to be able to forward the packets that it receives without having to unmarshal and then remarshal the arguments.

The effect of performing a group invocation using a group handle, is that a RPC buffer is constructed in the client which is passed to the RPC layer. This buffer has fields which instruct the RPC layer that it should, rather than performing a normal RPC, call a special purpose operation on the Group Leader, passing the RPC buffer as an argument.

The Group Leader's stub does not unmarshall the RPC buffer that it receives. Instead it prepares the buffer for being forwarded to all the Group Representatives for the group (including itself, but this can be optimized). The leader's stub then passes a list of the Group Representatives along with the RPC buffer to the RPC layer.

The RPC layer spawns a thread for each Group Representative. Each thread executes a normal RPC that passes the forwarding buffer to each Group Representative. This is depicted in Figure 8.

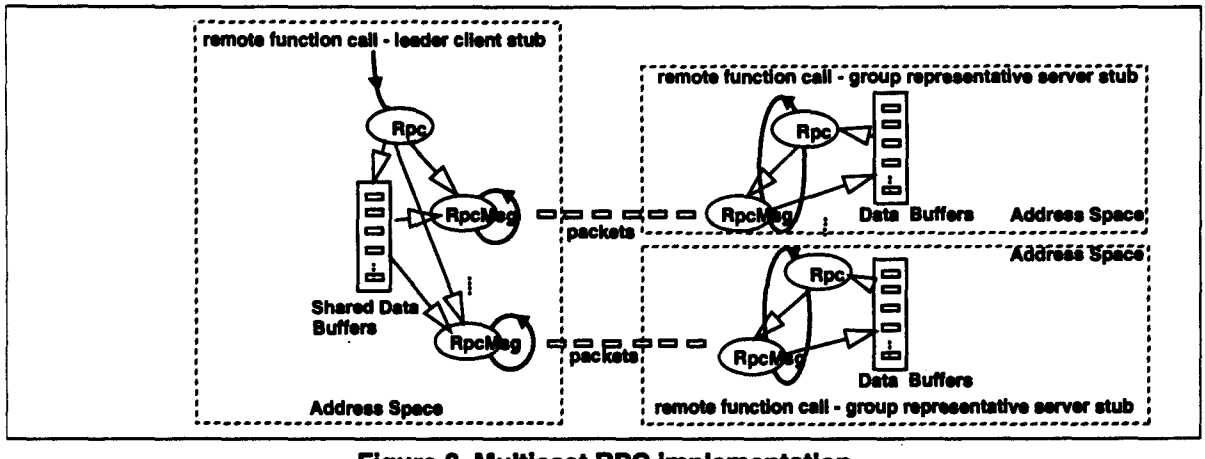

Figure 8. Multicast RPC implementation

This Figure illustrates the threads being spawned in the leader (each thread executes an `RpcMsg` object), and how each spawned thread does an RPC to each address space in which there is a Group Representative for the group that the leader represents.

When a Group Representative receives an invocation, the forwarded buffer is first unmarshalled, then for each member which is represented by the Group Representative, a thread is spawned which invokes the member via a local procedure call.

This protocol is extended to support multicast communication by requiring that acknowledgments from all receivers are received before the next burst is sent. If a receiver fails to send an acknowledgment, packets are retransmitted until the receiver either replies or is determined to have failed. Although this extension is relatively simple, it does have the drawback that multicast performance is limited by the slowest receiver.

10.3 Group Management

Group management is implemented mainly in terms of hash tables which maps group names to list of remote handles, and dynamic arrays of remote handles.

The Group Manager's hash table hashes group names to lists of handles to Group Representatives the first of which is the Group Leader. The Group Manager has operations for adding and removing handles of representatives, and for returning a handle to the leader of a group. The Group Manager also has an operation which allows its contents to be listed.

A Group Representative has a dynamic length array which contains the handles to the Group Members which it represents. A representative has operations which enables Group Members to request that their handles are added (join) or removed (leave) from the array of handles. A Group Representative also has operations for iteratively returning handles to its Group members. This is used to enable the server group stub (see 9.2) to get hold of handles to the members, so they can be invoked when a group invocation has been forwarded to the representative.

In addition to containing an array of handles for Group Members, the Group Leader also contains a list of handles for Group Representatives; this list is a cache of the list which is maintained in the Group Manager.

10.4 Concurrency control

The Group Manager can only process one operation invocation at a time. This is achieved by Extended-C++ `regions` in the implementation of each operation. A `region` prevents more than one of the Group Manager's threads to enter it at a time.

The Group Leader can only process one operation invocation at a time. This is achieved by a region in the implementation of each operation, which only allows one of the leader's threads to be in any region at a time.

A Group Representative can receive group invocations from the leader, and invocations that request that an Extended-C++ object is joined to the group. A representative prevents interference between these two kinds of invocations by way of a region in each of the operations that can be invoked. Only one of the representative's threads can be in any region at a time.

However, the Group Representative which is the leader of a group, will send group invocations to all representatives, in particular, also to itself. To prevent a deadlock from occurring in the leader while it is multicasting a group invocation which it is blocked from processing, the regions in a Group Representative allows one of the leader threads in a region at any time *and* one Group Representative thread in any region at a time. Because the leader is blocked while it multicasts to the representatives, the blocked leader thread cannot interfere with the Group Representative thread.

10.5 Performance figures

Some encouraging performance figures have been measured for the prototype on a HP730 (80 MIPS) machine running HPUX9.0. The average time for invoking a group from an EC++ object which resides in an address space on the same machine

as a varying number of members each of which execute in separate address spaces, are shown in table 1.

number of multicasts to group	number of members in group	average time in seconds for making a multicast	average multicast time in seconds per. member
1000	1	0.005	0.0050
1000	2	0.009	0.0045
1000	10	0.031	0.0031

Table 1: Extended-C++ multicast performance

For comparison, the average time for a RPC between two objects in different address spaces on the same machine (measured over 1000 RPCs) is 0.003 seconds. As seen from the last column in Table 1, the overhead of invoking a group is spread across the members of a group, so that the average time for a multicast to reach a member of a group converges towards the time it takes to make an RPC. Note, that these figures are for invocations between address spaces on the same machine. Also note that the buffer which is sent in the multicast contains only a single packet.

Performance figures for two other systems are provided below to give an idea of how our figures are positioned. Direct comparison of the figures should be avoided as different conditions apply to each set of figures. The performance figures that have been measured for ISISv3.0 [ISIS 92] on HP720 (60 MIPS) machines over ethernet are shown in Table 2. The ISIS figures are for 1Kbyte packets which are sent asynchronously and received in total order in the callees; the measured time is for when the first result, as opposed to all results as in our figures, have been returned to the caller.

number of receiving address spaces	time in seconds for receiving the first multicast reply	average multicast time per member in seconds for receiving the first multicast reply
1	0.0090	0.00900
2	0.0105	0.00525
3	0.0115	0.00383
4	0.017	0.00425

Table 2: ISIS abcast performance

The only available performance figures for the GEX protocol which supports groups in ANSAware [Oskiewicz 92] is that a multicast invocation of a group with two members on a HP425T (20 MIPS) machine takes 0.071 seconds. This figure includes an overhead caused by the generation of debugging information.

11 Related work

The conceptual design of multicast RPC in Extended-C++ is based on ideas from ANSA [Olsen 91]. ANSA groups are interfaces which are indistinguishable from singleton interfaces. The ANSA architecture for groups is much more general than the architecture described in this report. ANSA's architecture covers issues such as result collection, collation of multiple results into a single result, collation of replica invocations into single invocations, and synchronization of replica state. The ANSA architecture for groups also generalizes the separation of mechanisms from policies to enable provision of most appropriate policy and default policies [Oskiewicz 93].

The architecture for engineering groups in CareVue is based on ideas from Amoeba in which one kernel among a set of machine kernels can have a sequencer enabled while the others are disabled. This idea gave rise to our notion of a group leader which acts as the sequencer of group invocations.

Our architecture deviates from most others by assuming that the logically centralized Group Manager service can be implemented as a robust service which is independent of the architecture. ANSA's GEX protocol is a contrasting example, where Chang and Maxemchuck's protocol [Chang 84] is the inspiration for a distributed algorithm for providing total ordering of messages, which is executed by the members of a group.

Our support for multicast RPC provides total ordering of messages sent to the group members. ISIS takes great effort in enabling the programmer to select the minimum ordering required for a given application by providing various protocols with differing ordering guarantees. By chosing one ordering option, we restrict generality so the way multicast RPC is supported in Extended-C++ is very simple.

12 Conclusion

This report has described how multicast RPC is supported in Extended-C++ by a few simple and easy to understand language extensions. The notion of a group of objects was introduced as a concept for programming multicast RPC. The engineering of an object group is realized by a software architecture in which the components are hierarchically organized. This organization serves to minimize the number of messages that constitutes a multicast, and to minimize the information that each component maintains. The software architecture therefore scales well.

The provision of support for groups in Extended-C++ is not just an academic exercise. Extended-C++ is a language used for production programming in which the introduction of groups is a welcome abstraction for building event based applications and/or applications which are composed of replicated software components. Therefore the approach to supporting groups is pragmatic: simplicity and efficiency is favoured to generality. The architecture deviates from other approaches by assuming the availability of a very reliable service which is called the Group Manager. The information that this service maintains is well-defined, so it is feasible to hand-craft this service in devoted address spaces and/or machines.

The reliability of the architecture was discussed and it was shown how it could be extended to support atomic multicast RPC.

A prototype which is a simplified implementation of the architecture for multicast RPC was described. The performance figures that have been measured are very encouraging. They compare well to available figures for ANSA and ISIS, though the latter two systems provide additional support for fault-tolerance that are not yet available in the prototype.

13 Acknowledgements

Thanks to Robert Cole, Nigel Edwards, and John O'Connell for reviewing an earlier version of this report. Thanks are also due Nigel Edwards and Paul Harry for making available the performance figures for ANSAware GEX and ISIS, respectively.

14 References

[ANSA 89] "*An Engineer's Introduction to the Architecture*", ANSA, Poseidon House, Castle Park, Cambridge, U.K., 1992.

[Birrell 84] Birrell, A., Nelson, B., "*Implementing Remote Procedure Calls*", ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984.

[Birman 91] Birman, K.P, "*The Process Group Approach to Reliable Distributed Computing*", July3, 1991. Available from Cornell University.

[Chang 84] Chang, J., Maxemchuck, N., "*Reliable Broadcast protocols*", ACM Trans. on Computer Systems, Vol. 2, No. 3, 1984.

[Cooper 90] Cooper, E., "*Programming Language Support for Multicast Communication in Distributed Systems*", Proceedings of the 10th International Conference on Distributed Computing Systems, 1990.

[ISIS 92] ISIS Reference Manual Version 3.0.

[Kaashoek 91] Kaashoek, M. F., and Tannenbaum, A. S., "*Group Communication in the Amoeba Distributed Operating System*", IEEE, 1991.

[Olsen 91] Olsen, M.H., Oskiewicz, E., Warne, J.P., "*A Model for Interface Groups*", Proceedings of the Tenth Symposium on Reliable Distributed Systems, IEEE, 1991.

[Oskiewicz 92] Oskiewicz, E., and Edwards, N.J., "*GEX Design Notes - revised*", ANSA Report RC.328.00, ANSA, Poseidon House, Castle Park, Cambridge, U.K., 1992.

[Oskiewicz 93] Oskiewicz, E., and Edwards, N.J., "A Model for Interface Groups", ANSA Architecture Report AR.002.01, ANSA, Poseidon House, Castle Park, Cambridge, U.K., 1992.

[Seliger 90] Seliger, R., "*Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling, and Garbage Collection*", Proceedings of the 1990 Usenix C++ Conference, 1990.