Encapsulating Plurality

Andrew P. Black and Mark P. Immel

This paper describes the *Gaggle*, a mechanism for grouping and naming objects in an object-oriented distributed system. Using Gaggles, client objects can access distributed replicated services without regard for the number of objects that provide the service. Gaggles are not themselves a replication mechanism; instead they enable programmers to construct their own replicated distributed services in whatever way is appropriate for the application at hand, and then to encapsulate the result.

From the point of view of a client, a Gaggle can be named and invoked exactly like an object. However, Gaggles can be used to represent individual objects, several ordinary objects, or even several other Gaggles. In this way they encapsulate plurality. If a Gaggle is used as an invokee, *one* of the objects that it represents is chosen (non-deterministically) to receive the invocation.

1. Introduction

When designing a distributed object-oriented system, particular choices must be made for the implementation of each object in the system, and for the location of each object. These decisions can have a profound effect on the performance of the system as a whole, and they are therefore likely to be revisited and changed as the system evolves.

A good programming language and development environment will make it easy to encapsulate system components so that the effects of any changes in their implementation are localized. This is true even in centralized environments. Distributed programming environments, of which Emerald will be used as an example, go a step further: they also encapsulate distribution [5]. That is, the syntax and semantics (modulo failures) of the basic computational step, the invocation of a named operation on a receiver object, are independent of the location of the receiver.

A particular choice must also be made for the *number* of objects used to implement a particular abstraction. This choice can have a profound effect on the availability and reliability of the system as a whole. However, ordinary objects do not provide a way of encapsulating such a choice. This paper introduces the *Gaggle*, a facility for "encapsulating plurality", i.e., for hiding from a client the *number* of objects that implement a service. Simply put, Gaggles provide a way of treating a plurality of objects as if they were a single object.

Authors' electronic mail addresses: black@crl.dec.com; immel@husc.harvard.edu.
Contact address: Dr A. P. Black, Digital Equipment Corporation, Cambridge Research Laboratory, One Kendall Square, Bldg. 700, Cambridge, Massachusetts 02139, U.S.A.
Mark Immel was supported by the Division of Applied Sciences of Harvard University while this work was undertaken.

The context of this paper is the Emerald programming language, but we believe that the concept of the Gaggle is equally applicable to other environments for the programming of distributed applications.

1.1. An Example

To clarify the problem caused by replicated services, consider a highly available file service. High availability is achieved by using multiple servers.

Suppose that there are three servers and that the quorum consensus algorithm is used to ensure consistency [8]. Any object that wishes to read a file must therefore know both the identities of the servers and the algorithm for reading a file. We will refer to this information as the access data (shown as the small circle labeled "A" in Figure 1.)

It is possible for every client to have a separate and independent copy of the access data. This has the advantages of simplicity and robustness; provided a quorum of servers is available, the client will be able to obtain service (see Figure 1a). However, this arrangement has serious maintainability and transparency problems. First, the client must treat replicated files in a different way from non-replicated files. Second, if the replication algorithm is changed, perhaps to the available copies algorithm [2], then every client must be found and modified to reflect that change. In a wide-area distributed system this is infeasible.

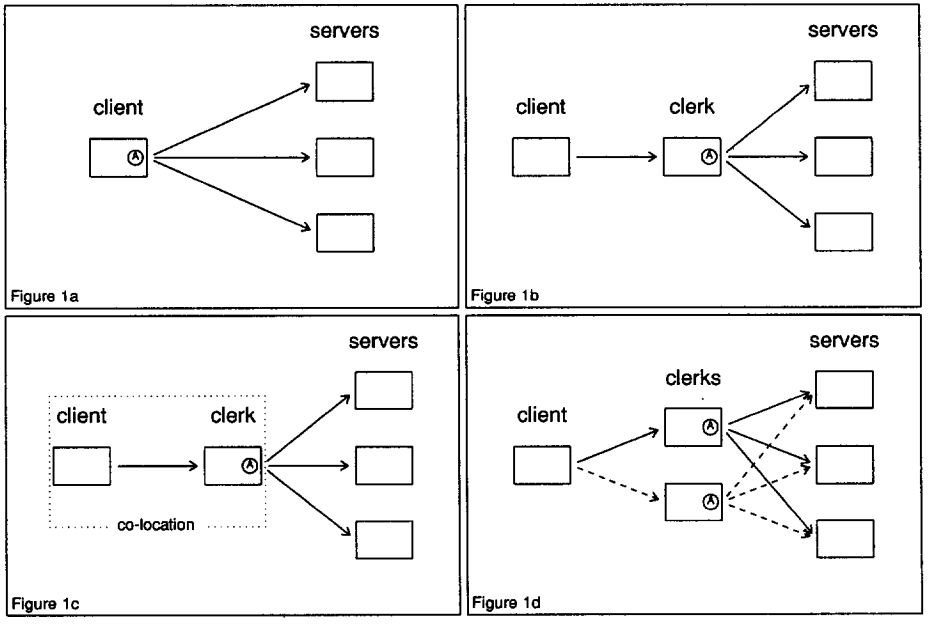

Figure 1: Four ways of accessing a replicated service.

The obvious solution to this problem is to encapsulate the access data in a clerk object, as shown in Figure 1b. When a client object needs to read a file, it invokes a clerk, which then executes the *read* algorithm on its behalf. Now only the clerk need be modified if the access data change. Moreover, if the same clerk interface is used to read non-replicated files, then the representation of a file can be changed from non-replicated to replicated without necessitating any changes in the client. Schroeder [13] discusses the Clerk paradigm in more detail.

Unfortunately, using a single clerk reintroduces a single point of failure into the system: if the clerk fails, then the client cannot read its file, even though the file servers may all be available. There are two ways to avoid this loss of availability: *co-location* of the clerk and the client, and *replication* of the clerk.

- If the client and the clerk are *co-located* in the same address space, the clerk is unlikely to fail separately from the client, and the existence of the clerk does not compromise availability. This arrangement is shown in Figure 1c, where the dotted line encloses the co-located objects. Although Emerald provides various facilities for requesting that objects be co-located, it is not always appropriate to solve the availability problem in this way.

  If the client is running in a small machine (perhaps a laptop computer), or if it obtains its services over a low-bandwidth link (perhaps a dial-up telephone line or a radio modem), then there are obvious load- and traffic-reduction reasons for locating the clerk near the servers. It is also advantageous to allow many clients to share a data cache kept by a single clerk; in this situation the clerk cannot be local to all of the clients. In addition, initializing a clerk may be time consuming, so that clients that expect to interact only briefly with the service will be best served by an existing clerk.

- An alternative is to *replicate* the clerk, and to allow clients to invoke any clerk that is available, as shown in Figure 1d. A client might pick a clerk essentially at random, or in a way that minimizes response time or that shares the load on the clerks. If the clerk used in the initial invocation is no longer available on a subsequent invocation, the client transparently "fails over" to one of the other clerks.

The Gaggle was conceived as a mechanism for transparently invoking one of a number of clerk objects. The problem that it solves is fitting the notion of transparent fail-over within a group of essentially equivalent clerks into an object-oriented computation. To understand this problem better, we must briefly look at the primitive mechanisms of such a computation.

1.2. Characteristics of Distributed Object-Oriented Computation

For the purposes of this paper, we can characterize distributed object-oriented computation by the collaboration of several active *objects* through the exchange of *invocation messages* and replies. Such computations proceed by means of invocations such as

buffer ← *file.read* [*offset*, *length*];

this invocation has four parts. *file* names some object, which we call the invokee; this is the object that will actually carry out the computation. *read* is the name of the operation that the invokee is requested to perform. [*offset*, *length*] is the argument list; the values of the arguments (i.e., the names of the objects that *offset* and *length* denote) are sent to the invokee along with the operation name, so that the invocation request can be parameterized. When the invocation completes, a result object has been generated, and the result variable *buffer* will name it.

In this computational model, the object that is to receive and execute the invocation cannot be distinguished from the object named in the invocation statement. In Emerald, objects are named using network unique identifiers, so the invokee *file* can be anywhere on the network. Moreover, *file* can move from one Emerald node to another: the invocation machinery will track it down. But Emerald does not enable one to express the idea that the invokee should be a plurality of objects.

1.3. Gaggles

A Gaggle behaves very much like an object: it can be named and invoked in the same way as an object. However, because a Gaggle can represent an individual object, several ordinary objects, or even several other Gaggles, a Gaggle encapsulates plurality. If a Gaggle is invoked, *one* of the objects that it represents is chosen to receive the invocation.

We designed Gaggles to be a flexible low-level tool, useful as a basis for experimentation, rather than as a finished solution to a particular problem. Their advantages are that they fit into our object-oriented computational model, including our rather rigid view of typing, and that the costs of the implementation fall only on those objects that use them.

The remainder of this paper is organized as follows. Section 2 discusses other work related to our proposal. Section 3 shows some of the ways in which Gaggles can be used, and illustrates their use in solving real problems of distributed computation. Section 4 looks at the design space for Gaggles, and discusses why we feel that our particular design choices are appropriate. Section 5 describes some possible implementations of Gaggles for systems of various scales. Section 6 summarizes the work and gives its current status.

2. Relationship with other work

2.1. Group Communication amongst Objects

Pardyak [12] has proposed a general model of group structure for the interaction of clients and replicated services in an object-oriented environment. Figure 2 shows his model; for consistency, we have re-labeled Pardyak's "group objects" as "clerk objects". The "member" objects are the servers that actually provide the replicated service; they can communicate amongst themselves using the member channel, or

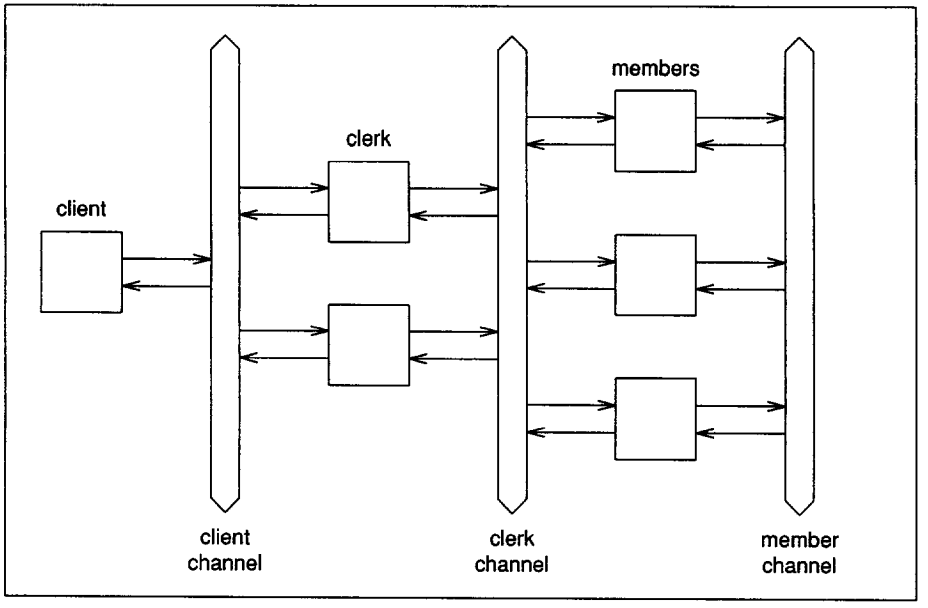

Figure 2: Pardyak's General Model of Group Structure (after [12])

with both members and clerks using the clerk channel. The clients access the service by using a client channel to communicate with several clerk objects.

Pardyak's implementation is less general than this model; it deals only with the special case in which there is a single clerk object. This restriction permits the client communications channel to be a traditional object invocation. Gaggles provide the tool to generalize his work so that multiple clerk objects are permitted. (Remarkably, we were not aware of this synergy until after the initial design of the Gaggle had been completed.)

2.2. ISIS Process Groups

ISIS, to quote its developers, is "a system for building applications consisting of cooperating, distributed processes. Group management and group communication are two basic building blocks provided by ISIS" [15]. The system has generated significant interest and is in use at several sites. ISIS presents users with a sharp set of tools for building one particular kind of fault-tolerant system, but it is not particularly useful if other kinds of system organization are preferred.

In the most recent formulation of ISIS [15, 16], the sender of a message must itself be a member of a group in order to send a message to that group. This means that an ISIS process group cannot replace an existing single server process directly. If a client wishes to make a request of a server group to which it does not belong, it must send the request to a particular member of the group called the contact, which will generally multicast the request to the whole group. If the contact fails, the client must somehow choose a new contact and and retry the request. Thus the

client must be aware of the fact that it is making use of a group of processes rather than a single process: plurality is not encapsulated.

The previous formulation of ISIS [3] had special support for clients of groups. However, this gave rise to significant implementation complexity, and was hard to make secure.

It should be emphasized that Gaggles do not compete with ISIS process groups; the two ideas are complementary. Gaggles provide encapsulation; process groups provide fault tolerance. Gaggles can be used to encapsulate any replication mechanism; ISIS process groups can be used with or without encapsulation.

2.3. Replicated Objects in Emerald

In existing Emerald systems, although great care is taken to present the illusion that each object has a unique representation, some objects are actually replicated by the implementation.

Emerald distinguishes immutable objects from mutable objects. Immutable objects are often small objects like integers, strings, or types; if a remote node wishes to access such an object, it makes sense to create a local copy on that node. Because the state of an immutable object does not change, the implementation can create many copies of an immutable object without affecting Emerald's shared object semantics.

User-defined objects can also be declared to be immutable; the implementation may choose to replicate such objects too. Emerald requires only that the *abstract* state of an immutable object not change; it is legal for an immutable object to change its *concrete* representation. For example, an immutable object that represents a function could memoize its results and still be immutable. If a programmer erroneously declares as **immutable** an object whose abstract state changes, then the result of a computation will depend on the number of copies that the implementation chooses to create.

Gaggles may be viewed as an extension of this mechanism. Invoking a Gaggle is like invoking an immutable object: one of the representative objects will be chosen to receive the invocation, and the semantics of the computation should be oblivious to this choice.

2.4. Fragmented Objects

The SOS system designed at INRIA [14] supports "Fragmented Objects"; fragments of a single object can be located on separate machines. To clients, the whole Fragmented Object appears to be a single object. However, each fragment can view the other fragments as if they are self-contained objects, and the failure of one fragment does not cause the failure of the whole Fragmented Object.

A client accesses a Fragmented Object through a fragment in its own address space, which plays a rôle similar to that of a stub in an RPC system. This stub fragment in turn accesses other fragments using a lower-level fragmented object called a channel. Any particular message is delivered to either a single fragment or

to multiple fragments, depending on the nature of the channel (which is under the control of the Fragmented Object). The channels and the fragments may run a membership protocol so that they are constantly aware of changes in availability. Before a client can use a Fragmented Object for the first time, it must go through a binding process to create an appropriate channel and encapsulate it in the stub fragment.

Compared to Gaggles, Fragmented Objects provide the programmer with much more flexibility, and associated complexity. provide one very simple way to invoke a plurality of objects; Fragmented Objects provide a collection of mechanism out of which the object designer can build his or her own robust invocation mechanism.

3. Using Gaggles

Gaggles are quite flexible, and can be used in many different ways. First we describe some of the usage paradigms or "idioms" that we have encountered; then we sketch some examples of more complete applications.

3.1. Usage Paradigms

A Gaggle of Independent Objects.

This is the most obvious way of using a Gaggle. A number of equivalent and independent objects are placed in the Gaggle; the client invokes the Gaggle directly for service. A Gaggle of independent objects might be used to access one of a number of equivalent time servers or name servers.

The Consistent Gaggle.

Although Gaggles do not themselves provide any consistency, they can be used to encapsulate the interface to a group of objects that do collaborate to maintain consistency. The replicated file service described in the introduction illustrates one way of achieving consistency. An ISIS-style process group is another way. Because the members of the Gaggle can refer to each other by their own names, any consistency algorithm of the implementor's choice can be used. Clients of the consistent Gaggle invoke it using the name of the Gaggle invokee.

A Gaggle of Workers.

A Gaggle of objects can be used to encapsulate a pool of worker processes that provide computational cycles for a client. Suppose a Gaggle of n objects is created and located on various hosts in order to distribute the computational load. A client object would invoke the Gaggle to initiate work on a particular subtask. Such a request would be received by an arbitrary member of the Gaggle. However, because each member of the Gaggle would be aware of the rule used to share tasks, it would be a trivial matter for the receiver to forward the task to the appropriate worker. For example, the convention might be to assign task k to worker object k **mod** n.

If the amount of data associated with the task is large, this arrangement has the disadvantage that the data must be sent to the receiver and then forwarded to the

worker. This can be avoided by not sending the data in the request message; instead the client's name is sent. Once the worker is selected, it can then invoke the client to request its share of the data directly.

Multiple Names per Object.

Gaggles can be used to provide multiple names (object identifiers) for a single object. The object simply creates several Gaggles, and then makes itself a member of each. Each Gaggle will have a distinct name. So long as it is the only member, all invocations on the Gaggles will arrive at the creating object.

To allow an object to have different behaviours when it is invoked by its various names, we have added a new keyword (**invokee**) that permits an object to ascertain the name used by the invoker (see section 4.3).

Hiding Object Identity.

It has long been assumed that because the *concept* of object identity is essential to describing the semantics of object-oriented languages, the ability to *test* the identity of two object references is similarly essential. Recently, this assumption has been questioned. The ANSA system [1] does not provide a built-in way of testing object identity at all; if the application demands the ability to test the identity of a certain class of objects, then this can be provided by adding an explicit *getIdentity* operation to the code that defines those objects.

In a system like Emerald that *does* provide testable object identity, Gaggles can be used to hide it. This is done by creating multiple names for an object, as described in the previous section. Indeed, the very *possibility* that an object might be known by multiple names means that programmers cannot assume that distinct names refer to distinct objects. Thus, if it is essential that a reliable identity test be available on a particular class of object, the programmer should provide those objects with a *getIdentity* operation.

3.2. Some Applications of Gaggles

Accessing a Name Service.

Lampson has described a highly available large scale name service [11] with many servers, each supporting some fragment of the naming tree. A given directory may be replicated on several servers. Each server also keeps information on how to access the servers that store the parents of its directories.

In order to resolve a name one typically accesses a local server. If that server stores the appropriate directory, it will return the value associated with the name in question. Otherwise, it will return information that will help the requester find an authoritative name server.

Updates are made in a similar way. Because there may be multiple copies of a single directory, and two updates to the same subtree may be received by different servers, it is possible for different servers to present inconsistent views of the namespace. Various algorithms are run on the servers to ensure that all changes will *eventually* be reflected at all concerned servers. However, clients may see temporary

inconsistencies, such as a newly-created sub-directory not yet being visible in an ennumeration of the parent directory.

A client of the name service typically uses an object (a clerk) to provide a single access point for the service. The clerk encapsulates information about the service, such as which servers support various parts of the namespace, and may also cache the results of previous queries. The name server clerk deals with an enquiry by first searching its own cache, and if possible returning a result without ever contacting a name server. If no match is found in the cache, the clerk makes a corresponding inquiry of a nearby name server. The clerk can keep information about the responsiveness and location of several servers to improve performance. Updates are first reflected in the cache, and then forwarded to a server for the appropriate naming domain.

The advantage of this arrangement is that while the clerk improves responsiveness, it keeps no vital data. There is no need for the clerk to be persistent. If it crashes, the client can use a new clerk. In fact, there typically will be several clerks active at a give time; it does not matter which one the client uses.

This sort of application can be implemented conveniently by making the clerks a Gaggle of independent objects. Using a sufficiently-large Gaggle helps to ensure that a clerk is always available. Since using the same clerk in successive operations is not necessary for correctness, the fact that successive Gaggle invocations may be dealt with by different receivers is not a concern.

Causally Consistent Name Service.

Although the weak consistency of a Lampson-style name service helps to ensure high-availability and low latency, Ladin has observed that sometimes this semantics is inadequate [10]. For example, suppose a system administrator wishes to create a new sub-directory and then to populate it with information about printers. If the update that creates the directory goes to one server and the request to add the first printer goes to another, the operation to add the printer may fail because the directory does not exist. In this situation, high availability doesn't help: the perceived behaviour does not meet the user's requirements.

Ladin describes a mechanism that lets the client obtain whatever degree of consistency it requires. Each update to the namespace returns an identifier. Enquiries and updates may require that the state on which they operate is "later" than the state created by a certain set of updates; this set is identified *explicitly* by providing the set of update identifiers as an argument to the name server request. Using this mechanism it is possible to state that the various additions to the printer directory must be "later" than the update that created the directory. If such an addition happened to be sent to a server that has not yet seen the creation of the directory (i.e., it arrives too "early"), the addition will be delayed until the server has been able to obtain and apply the update that created the directory. In this way the algorithm achieves tighter consistency, possibly at the expense of increased response times.

A Gaggle of independent objects can be used to provide an interface to a Ladin-style name service just as effectively as to a Lampson-style name service, and in the same way. However, in a small-scale Ladin-style name service, where all directories are fully replicated, the rôle of the clerk in finding an authoritative server for a given operation disappears. Any server will do. In this situation, all the servers can be made members of a consistent Gaggle that can *itself* be the receiver of the updates and queries.

Mail dispatch.

When sending mail, the user agent needs to contact a message transfer agent that is willing to store and forward its messages. Any of a number of message transfer agents will serve.

This problem can be solved by constructing a Gaggle of independent message transfer agents, and by invoking the Gaggle to deposit outgoing mail. This Gaggle invocation will succeed if any one of the transfer agents can be found.

One Object, Many Rôles.

Sometimes a single object may play several rôles. For example, in a hierarchic file system the rôles of the various directory objects might all be implemented by a single B-tree object, rather than by having a separate object for each directory.

Reference 4 discusses a similar situation: a file that can be read by multiple clients simultaneously. A current file position index must be kept for each reader. The file object might create a channel sub-object for each reader; in this case the current file position would be implicit. Alternatively, the file might service all of the readers itself, and require that each read request supply a channel identifier.

A system designer ought to be able to use a single object playing multiple rôles, or to use a separate object for each rôle, depending on the costs and constraints of a particular situation. It might even be desirable to try both design alternatives. Unfortunately, in an object-oriented system in which each object has a single name, the two alternatives present different interfaces to the client. For example, if the operation to read from a channel relies on the identity of the channel object to determine which stream is to be read, then it would not be possible to combine several streams into a single multi-channel object, since an additional parameter would be required to select the required stream.

As we showed in Section 3.1, Gaggles provide a mechanism for giving an object multiple names. This enables each client to invoke a receiver specific to its channel, while still allowing the implementor to choose between the above strategies. A single file object known by multiple names could use the value of the invokee to choose the correct channel. Alternatively, the multiple names could be used to refer to genuinely distinct open file objects.

Type Restriction.

Hutchinson has experimented with an extension to Emerald in which it is possible to

restrict the dynamic type of an object†. In this extension, the statement

restrict *e* **to** *t*

denotes a new object that has the same value as *e* but whose type is *t*. It is legal whenever **typeof** *e* – the dynamic type of the object denoted by *e* – conforms to *t*. This means that **typeof** *e* must be more general than *t*; *e* contains more operations, or the operations return more general results. Type restriction is not part of the standard Emerald language, but the same functionality can be obtained using Gaggles.

In Emerald, the dynamic type of an object will always conform to the syntactic type of any expression that evaluates to it. If $\mathcal{E}[\![expr]\!]$ denotes the value of *expr*, and $\mathcal{T}[\![expr]\!]$ denotes the *syntactic* type of *expr*, we have

$$\mathcal{E}[\![\textbf{typeof } n]\!] \geq \mathcal{T}[\![n]\!] .$$

The Emerald type checker guarantees this invariant, and also ensures that for every invocation *n.op*, the operation *op* is contained in the type $\mathcal{T}[\![n]\!]$ [7]. This means that if *n* is a variable that names an object *b*, only a subset of *b*'s operations will in general be available for invocation on *n*.

It is tempting to use this mechanism to restrict the operations that certain clients may perform on a particular object. However, such a restriction can always be circumvented, because an explicit widening coercion (a **view** expression) can be used to transform *n* into an expression with wider type on which all of *b*'s operation can be invoked.

However, Gaggles can be used to achieve the effect of secure narrowing. Consider a Gaggle invokee *B* whose type is a restriction of the dynamic type of an object *b*, and which has *b* as its only member. If the **typeof** primitive is applied to *B*, the result will be the restricted type (as explained in section 4.2), and any attempt to **view** it as a wider type will fail. Any invocations sent to *B* will be received by *b*, since it is the only member of the Gaggle.

4. The Design of Gaggles

This section presents the current design of Gaggles and discusses some of the alternatives we considered.

4.1. Syntax

One of the last issues that we dealt with is how Gaggles should appear in the Emerald language. In some ways, Gaggles are similar to other forms of collection, like Array and Vector, which are already in the language. Constructors for these collections are presented as Emerald objects. Much of their implementation is also in Emerald, although some operations must be implemented by system primitives. It seemed desirable to present Gaggles in the same manner. This approach minimizes changes to the language itself; the only additions are system primitives.

† Norman Hutchinson, Personal Communication.

The object *Gaggle* is similar to *Array*; it is immutable and has the following interface:

function *of* [*AType* : *Type*] → [*result* : *GaggleType*] .

The result of *Gaggle.of*[*AType*] is an immutable object with the following interface (which we will call *GaggleType*)

function *getSignature* [ ] → [*Signature*]
operation *new* [ ] → [*GaggleManager*] .

There are two ways of viewing a Gaggle. A *GaggleManager* represents the management interface of the Gaggle, and exports operations to add members and to access the service interface:

operation *addMember* [*AType*] → [ ] *% the only management operation*
function *invokee* [ ] → [*AType*] *% returns the service interface* .

The result of *invokee* looks like an ordinary object of the type given in the *of* operation. In particular, it can be invoked; for this reason we call this entity the GaggleInvokee.

The following code fragment shows how a Gaggle might be populated and used.

const *aGaggleManager* ← *Gaggle.of*[*NSClerk*].*new*
aGaggleManager.addMember[*NS.lookup*["*primary clerk*"]]
aGaggleManager.addMember[*NS.lookup*["*alternate clerk*"]]
aGaggleManager.addMember[*NS.lookup*["*backup clerk*"]]
const *aClerk* : *NSClerk* ← *aGaggleManager.invokee*[ ]

After this sequence, *aClerk* is bound to the Gaggle's service interface, which can be treated like an object of type *NSClerk*.

4.2. Semantics

In order to make a GaggleInvokee like an object, every aspect of the language that involves objects must be defined for GaggleInvokees.

In Emerald, a program may **move** an object to a location, **fix** an object at a location, **unfix** an object, **refix** an object at a new location, determine the **typeof** an object, **view** an object as having another type, determine whether an object **isfixed**, invoke an object, and **locate** an object. We have devised ways of handling all of these primitives.

- The Emerald **move** primitive is actually a hint; the implementation is not required to perform the move suggested. Thus, one alternative is to say that move applied to a GaggleInvokee does nothing. But, it is conceivable that the Gaggle would like to take some action when an object tries to move it, such as creating a new member at the specified location. For this reason, members of a Gaggle may provide an operation *move_handler* which takes the appropriate action.

The semantics for move, when applied to a GaggleInvokee, is to invoke the operation *move_handler* on the GaggleInvokee. If the chosen receiver does not have such an operation, no action is taken.

- The **fix** statement fixes the location of an object; this prevents the object from moving until it is unfixed. Often, rather than specifying a location by means of a node object, the programmer specifies a second object at which to fix the first. In this case, the location of the second object is ascertained, and the first object is fixed at the same location. The **fix** statement fails if the object is already fixed.

  Since immutable objects are copied rather than moved, fixing an immutable object is a null operation. The location of an immutable object is always "here" (it is co-located with the enquirer), so fixing an object **at** an immutable object has the same effect as fixing an object at the current location. (At one time we considered creating a special object **everywhere** to represent the location of immutable objects; however, this was never implemented.)

  We considered imitating these semantics for Gaggles, but decided against it. Because the number and location of the members of a Gaggle are chosen by its manager, not by the system, there is no guarantee that there will always be a member at a particular location, and it seemed inappropriate to allow the statement **fix** *g* **at** *l* to succeed in spite of the fact that no member of the Gaggle *g* is at location *l*.

  The same objection can be raised against another alternative: treating **fix** like **move**, i.e., to say that members of a Gaggle may provide an operation *fix_handler*. Since **move** is a hint, the (user-defined) implementation cannot be "wrong". But **fix**, unlike **move**, is not a hint.

  We finally decided to make it an error to **fix** a GaggleInvokee, or to fix an object at a GaggleInvokee. With hindsight, it might also be wise for it to be an error to fix an immutable object.

- Since it is not an error to **unfix** an object which is not currently fixed, it is not an error to **unfix** a GaggleInvokee.

- The **refix** construct is an atomic **unfix** and **fix**; like **fix** it fails if either the object or location is a GaggleInvokee.

- The **typeof** a GaggleInvokee is the type that was used as argument to the *Gaggle.of* [···] invocation that built the Gaggle manager. **typeof** does not return the type of any particular member, which might contain more operations). This is because the meaning of the statement "object *o* has type *T*" is that all the operations in *T* can be invoked on *o* without danger of a "Message-Not-Understood" error. In the case of a Gaggle, we can offer this guarantee only for the type specified as the argument to *Gaggle.of* [···].

- The expression **view** *o* **as** *t* has syntactic type *t*. At execution time, the system checks that **typeof** *o* $\geqslant$ *t*. If it does, the view expression succeeds (and returns the *o*); otherwise, the view expression fails. This rule needs no modification

for GaggleInvokees.

- The **isfixed** predicate tests whether or not a particular object is fixed. Since a GaggleInvokee cannot be fixed, **isfixed** applied to a GaggleInvokee is always false.
- Invocation of a GaggleInvokee is defined to be the invocation of some member of the Gaggle, if a member can be found. If no member can be found with reasonable effort, the GaggleInvokee is considered unavailable. The implementation is free to invoke any member, and need not choose the same object on consecutive invocations. Our reasons for this choice are discussed in Section 4.4.
- The expression **locate** *o* returns the current location of the object *o*. Evaluation of a **locate** expression requires running the object-finding algorithm normally used for invocation; **nil** is returned if the object cannot be found. For GaggleInvokees, an attempt is made to find some member of the Gaggle, and return its location. If no member can be found with reasonable effort, **nil** is returned.

  At first glance, this may seem like a rather peculiar semantics for **locate**. Two consecutive **locate** statements could return locations on different continents. But the same possibility exists without Gaggles, because objects are mobile.

4.3. Other Language Constructs

Emerald contains one other primitive, **self**, that must be clarified in the presence of Gaggles. In addition, we have added two new primitives, denoted by the keywords **isplural** and **invokee**.

- **self** always denotes the current object, i.e., the one that evaluates **self**. In the case of a member of a Gaggle, it does not refer to the Gaggle but to the member itself.
- **invokee** denotes the name of the current invokee. An object may need to know if it has been invoked as a member of a Gaggle (and if so, of which Gaggle) or if it is has been invoked under its own name. **invokee** returns the GaggleInvokee if the object was invoked as part of a Gaggle, and **self** otherwise.

  This keyword is permitted only within the bodies of operations. Inside an initially, recovery, or process section the object has not been invoked and thus **invokee** has no meaning.
- **isplural** is a primitive predicate; **isplural** *o* returns **true** if *o* is a GaggleInvokee, and **false** otherwise. Although a client cannot violate the encapsulation of plurality and see the members of a Gaggle, it may occasionally be necessary to know whether or not an object is a GaggleInvokee. The provision of **isplural** is in the same spirit as the provision of **locate**; we expect that it will be used infrequently, but that sometimes it will be required.

4.4. Semantics of Gaggle Invocation

Like objects, GaggleInvokees can be invoked. When this happens, the invocation could conceivably be sent to one, some, or all of the members of the Gaggle.

Sending the invocation to all members is an unwise choice, because if one member is unavailable the invocation must fail, and one of our goals was to increase availability. A more reasonable choice is to send the invocation to all *available* members; this is the option chosen by the ISIS process group mechanism. However, this means that the implementation must keep a list of currently available members, and that the delivery of invocations must be consistent with this list. In other words, it requires a full implementation of causally consistent groups. Our intention is that the Gaggle should be a lighter-weight mechanism that can be used to *build* causally consistent groups.

Another possibility is to invoke *many* of the members: not necessarily all, but several. This is not very useful; consistency cannot be achieved by this mechanism alone, and having all members that receive the invocation communicate it to all other members would generate an unnecessarily large number of messages.

The remaining alternative is to invoke a single member object. There are motivations for this choice besides the process of elimination: this is all that many applications require (see Section 3), and the other invocation protocols can be built on top of this mechanism. This alternative maintains the property that the users of a service are the only ones who pay for them. In addition, it results in simpler semantics and implementation.

4.5. Failure Semantics

With the above semantics for invocation, one should not interpret the fact that the GaggleInvokee is unavailable to indicate that every member of the Gaggle is broken. Indeed, given two simultaneous invocations of the same GaggleInvokee, one may fail while the other succeeds.

We have considered allowing an object to retry a failed invocation, while indicating to the system that it should try harder. Objects could thus try increasingly expensive levels of invocation before deciding that the GaggleInvokee is unavailable. Another possibility is to allow invoking objects to indicate the "permissible expense" of an invocation directly. Then, the implementation would try no harder than instructed to locate the object.

4.6. Ownership vs. Multiple Interfaces

We explored various alternatives for providing the management and service interfaces before settling on the scheme described above. One alternative was that one member of the Gaggle would be special: it would be the "owner". execute management operations. Unfortunately, if the owner crashed, the Gaggle would be incapacitated. We therefore considered allowing the owner to pass ownership rights to other objects.

The ownership concept implied that the owner of a Gaggle would not have the same type as other members of the Gaggle, since it would have additional operations for management. What, then, would be the type of a GaggleInvokee? This problem prompted us to conceive our present solution, which uses two distinct object identifiers to name the two distinct interfaces.

4.7. Types

The type of each member of a Gaggle must conform to the type of the GaggleInvokee. This does not imply that each member must have the same concrete type, or even the same abstract type. This typing rule is the most lenient possible: it is the minimum requirement on the members that ensures that sending an invocation to any one of the members will not result in a "Message-Not-Understood" error.

4.8. Removal and Enumeration

We do not provide these services for Gaggles. To do so would require maintaining a list of all members of a Gaggle and running consistency protocols. The price we pay for our light-weight Gaggles is that these operations are not possible. However, if they are required, these operations can be implemented at the user-level as follows.

An object can forward invocations to another object by making the body of each of its operations invoke the corresponding operation on the other object. Similarly, it may forward invocations to a set of objects by maintaining a list of the objects and forwarding invocations to each. The forwarding object can provide a list of the objects to which it currently forwards; these are the members. If it accepts instructions to update the list, it can also export operations to add and remove members.

However, the forwarding object is a single point of failure. If it breaks, the entire group is unavailable. This can be remedied by making the forwarding object a Gaggle. The Gaggle can also provide the desired removal and enumeration facilities; the members of the Gaggle run their own consistency protocol to ensure that membership changes are seen by all members of the Gaggle. Users of the group invoke the Gaggle, which forwards the invocation to the members of the group.

4.9. Gaggles as members of Gaggles

Since Gaggles are designed to be treated as objects, there is no reason why Gaggles should not be members of other Gaggles. An invocation of a Gaggle can be forwarded to any member, including a member that is a Gaggle. The consequence is that the invocation must eventually be delivered to a member of the transitive closure of the Gaggle that was initially invoked. Since Gaggles have no membership list, we cannot prevent a Gaggle from being (indirectly) a member of itself. This implies that the invocation protocol must detect cycles.

5. Implementation Considerations

Although do not yet have a complete implementation of Gaggles, we have considered many possible implementation strategies and their suitability for systems of varying scale. We are pleased that the design of Gaggles has not made implementation inflexible; indeed, we often found ourselves dazzled by the array of alternatives. Just as it is hard to imagine a single object finding algorithm suitable for systems of vastly differing scale, it is unlikely that a single Gaggle invocation algorithm will suffice in all situations. Given some basic constraints, such as the inclusion of protocol version numbers in the headers of messages, there is no reason why the implementation of Gaggles could not be different on different nodes, or be changed dynamically while the system is running.

There are two major primitives to be implemented: adding a member to a Gaggle, and invoking a gaggleInvokee. It seems that there is a tradeoff between work done at the time a member is added and at the time an invocation is made. At one extreme, we could tell every node in the network about every new member. Then, every node would have a complete membership list and invocation would easy. At the other extreme, we could tell no other node about the new member, and to perform an invocation we could ask every node if it knows of any member of the Gaggle.

There is no parallel to the Gaggle *addMember* operation in an ordinary Emerald system. However, we do have experience with various algorithms for finding (singular) objects. An understanding of these algorithms will form the basis on which we can build a Gaggle finding algorithm.

5.1. Algorithms for Finding Objects

Emerald combines the process of finding an object with the process of invoking it. This is done for efficiency (objects are usually found at their previous location) and for correctness (an object might move between the two stages of an algorithms that first found an object and then invoked it).

The Broadcast Algorithm.

The original Emerald system ran over a five-node local area network at the University of Washington. The identities of the nodes were static and well-known.

If a node needed to invoke an object not present locally, it checked to see if it had a forwarding pointer to the object (a last known location of the object). If it did, the invocation was sent to that node. If the object was still there, the invocation would succeed. Otherwise, the forwarding chain would be followed until either the object was found and invoked or the chain broke.

If the object was found, the invoking node received a new forwarding pointer so that it could update its local tables. If the invoking node had no initial forwarding pointer, or if the forwarding chain had broken, the invoking node used broadcasts and, if they failed, a series of reliable point-to-point messages to all other nodes in the network. The object would either be found, or it would be determined to be unavailable.

Although this algorithm was suitable for a small scale local area area network, it is clear that it will not scale to the wide area.

The Hermes Algorithm.

The Hermes location algorithm was designed for the wide area and does not require broadcasts [6]. Forwarding pointers are still used, but in the event of a broken forwarding chain, the location of the object is obtained from stable storage.

Each Hermes object has a current location and a storesite, both of which may change. The storesite is a stable storage device that preserves a record of the object's state. When an object moves, the node from which the object is moving keeps a forwarding pointer to the new location; in addition, the storesite is informed of the new location. The forwarding pointers are appropriately timestamped so that one can tell which of two forwarding pointers is newer. Whenever a reference to an object is passed from one node to another, a pointer to its last known location is passed as well. Thus, when one object invokes another, the Hermes algorithm always has a forwarding pointer to follow. If the chain of forwarding pointers is broken, the location of the object is retrieved from stable storage.

Since the object can change its storesite, finding the current storesite is not trivial. The name of the object's initial storesite is encoded in its identifier. When the object chooses a new storesite, the old storesite is required to keep a forwarding pointer to the new one; the name service is also informed. When the location of the new storesite has become stable in the name service, the old storesite can forget the forwarding pointer.

Modifying the finding algorithms for Gaggles.

It is clear that following forwarding pointers will not work one object identifier can refer to many objects. This is precisely the situation we have created with Gaggles.

One solution is to break the problem into two pieces. First, when a node invokes a gaggleInvokee, it selects a particular member to invoke. Then the Hermes algorithm is used to find and invoke that member. Unfortunately, this gives up the advantages of the Gaggle's non-deterministic invocation semantics. If the particular member chosen happens to be at the end of a long forwarding chain, the invocation will be slow, even though some of the nodes that participated in the forwarding process might host other members of the Gaggle.

The alternative approach – invoking all known members in parallel – violates the semantics of Gaggles. Finding all known members in parallel and then invoking the one that is found first is correct but potentially expensive. We are forced to see a compromise.

If a member of the invoked Gaggle is local to the invoking node, the invocation can be performed without further ado. If there is no local member, one of the known members is selected (using the best information that is available locally) and the forwarding chain for that object is followed. However, the invocation message contains not only the identifier of the selected object, but also a tag indicating that this is a Gaggle invocation, and the identifier of the

gaggleInvokee. Now the recipient of this message has more freedom of action. If the selected member of the Gaggle is local, the invocation can be performed. If it is not, but some other member of the Gaggle is local, the invocation can *still* be performed. Failing these happy eventualities, the recipient can either follow the forwarding chain for the initially selected object, or it can substitute some other known member of the Gaggle and forward the invocation to it.

Some care must be taken to ensure that this algorithm terminates. It is sufficient to record in the invocation message the most recent timestamped forwarding pointers to all of the members on which invocation has been attempted. In the case of nested Gaggles, this will also serve to detect membership cycles.

The effectiveness of this algorithm depends on the care with which we select the particular member to which the invocation is forwarded at each step. The selection is assisted by keeping as much information as possible about the various members. In addition to timestamped forwarding pointers, we might keep information about network topology, system loads, and response times measured on previous invocations.

The initial invoker (indeed, any node on the invocation path) can also limit the flexibility that it grants to other nodes later in the chain by setting a maximum number of forwarding steps; when the maximum is reached, the invocation is forwarded no further, but instead a progress report is returned to the initial invoker containing updated membership information and forwarding pointers. The initial invoker can then decide whether to continue with its first choice of member or to try a different member. (A similar facility appears in the Hermes algorithm under the name of a hop-count; however, since in Hermes the invoked object is singular, the initial invoker has no freedom of choice. However, limiting the length of forwarding chains does increase robustness.)

In general, there are two parameters for the location algorithm: the topology of the network, and the members of the Gaggle. Given complete information about both, it would be easy to select the best member to invoke. But we do not have such information. Our response is twofold: first, we try to propagate as much information as we can, as cheaply as possible; second, we recognize that our information will never be complete, and strive do as well as possible with what we have.

Stable Storage.

The Hermes object-finding algorithm uses stable storage as a last resort. We can increase the robustness of Gaggles by using the name service to provide a form of stable storage for the gaggleManager.

Whenever a member is added to a group, it is possible to immediately update the (global) name-service. Alternatively, the updates can be batched. Now, if a node cannot find those members of a Gaggle that it knows about, it can ask the name server for a membership list and see if there are additional members. Since the name server is not up-to-date, it cannot be relied on to supply the names of all members; since it is a global service, it is relatively expensive, and should be used

only as a last resort. However, since members are never removed from a Gaggle, the information received from the name service, while incomplete, will never be incorrect.

Thus, stable storage for a Gaggle is implemented a little differently than for an ordinary object, but not inelegantly. Just as objects that exist in a single place store their state at a single storesite, objects that exist in a plurality of places store their state in a "distributed storesite": a name service.

The *addMember* operation.

When a member is added to a Gaggle, it is not clear to whom that information should be propagated. The membership table on the node where the *addMember* invocation occurred should certainly be updated. In addition, a name server update might be made or queued, as mentioned above. Beyond this, various alternatives are possible.

The reason to propagate membership information is to reduce the message traffic necessary to implement an invocation. Any propagation strategy that uses extra messages is therefore suspect. However, piggybacking Gaggle membership information onto existing node to node communication is promising. For example, nearby machines could be notified of new Gaggle members when status and load information are exchanged. Or, the next time an invocation for the gaggleInvokee is seen by the adding node, the membership update could be returned.

5.2. The Emerald implementation of Gaggles

The code for the Emerald run-time library that implements Gaggles is given in Figure 3. The only necessary additions to the Emerald language are the system primitives that implement *addMember* and that generate a new name (an object identifier) for the gaggleInvokee. This name is the only state in a gaggleManager; it is constant and assigned at object creation time, so the gaggleManager is immutable. Thus, when the name of agaggleManager is passed from one object to another, giving another object the ability to manage the Gaggle, the gaggleManager can be passed by value. Thus, if the name of the gaggleManager is passed to several objects on different nodes, the gaggleManager has been automatically replicated.

6. Current Status

The original implementation of Emerald generated highly efficient native code for a network of microVAX™ II workstations [9]. It has been ported to networks of SUN™ workstations. However, both the generation of native code and the history of the implementation have made the compiler and run time support for this version of the system hard to maintain.

™ VAX is a trademark of Digital Equipment Corporation. SUN is a trademark of SUN microsystems, Inc.

```
const Gaggle ←
  immutable object Gaggle
    export function of [memberType : type ] → [result : gaggleType ]
      where gaggleType ←
        typeobject gaggleType
          function getSignature [ ] → [Signature]
          operation new [ ] → [gaggleManager]
        end gaggleType
      where gaggleManager ←
        typeobject gaggleManager
          operation addMember [memberType ] → [ ]
          function invokee [ ] → [memberType ]
        end gaggleManager
      result ←
        immutable object gaggleCons
          export function getSignature [ ] → [s : Signature]
            s ← gaggleManager
          end getSignature
          export operation new [ ] → [aGaggleManager : gaggleManager ]
            aGaggleManager ←
              immutable object manager
                initially
                  const theInvokee ← % a system primitive generating
                                     % a new object identifier
                end initially
                export operation addMember [newMember : memberType ] → [ ]
                  % A system primitive handling member addition
                end addMember
                export operation invokee [ ] → [gaggleInvokee : memberType ]
                  gaggleInvokee ← theInvokee
                end invokee
              end manager
          end new
        end GaggleCons
  end Gaggle
```

Figure 3: Emerald implementation of Gaggles

To promote the use of Emerald as a teaching and research tool, Norman Hutchison created a portable version of the Emerald compiler (written in Emerald), and a portable run-time system (written in portable C). However, this version of the language was restricted to a single address space.

During the summer of 1992, the present authors started to add distribution to the portable version of Emerald. As this work progressed, we began to consider the extensions described here. We hope that Gaggles will be implemented as part of our ongoing work to complete the distributed portable implementation of Emerald.

We have not come to any definite conclusion about the relative merits of one Gaggle finding algorithm over another; only benchmarks run on real implementations can definitively decide that question. However, we have

highlighted some of the problems involved, examined many of the choices, and presented some plausible solutions.

Acknowledgements

We wish to thank the Cambridge Research Laboratory of Digital Equipment Corporation for providing computing resources and for the support of the first author, and the Division of Applied Sciences of Harvard University for the support of the second author while the work reported here was undertaken. We also wish to thank Norman Hutchinson and Eric Jul for helpful discussions while Gaggles were being designed, Robbert van Renesse for his assistance in obtaining information about ISIS, and Marc Shapiro and Messac Makpangou for information about SOS.

References

[1] Architecture Projects Management Ltd. "ANSA: An Application Programmer's Introduction to the Architecture". TR.017, APM Ltd, November 1991.

[2] Bernstein, P. A. and Goodman, N. "An Algorithm for Concurrency Control and Recovery in Replicated Distributed Databases". *Trans. Database Systems* **9**, *4* (December 1984), pp.596-615.

[3] Birman, K. P., Schiper, A. and Stephenson, P. "Lightweight Causal and Atomic Group Multicast". *Trans. Computer Systems* **9**, *3* (August 1991), pp.272-314.

[4] Black, A. P. "Supporting Distributed Applications: Experience with Eden". *Proc. 10th ACM Symp. on Operating Systems Prin.*, December 1985, pp.181-193.

[5] Black, A. P., Hutchinson, N., Jul, E., Levy, H. M. and Carter, L. "Distribution and Abstract Types in Emerald". *IEEE Trans. on Software Eng.* **SE-13**, *1* (January 1987), pp.65-76.

[6] Black, A. P. and Artsy, Y. "Implementing Location Independent Invocation". *IEEE Trans. on Parallel and Distributed Syst.* **1**, *1* (January 1990), pp.107-119.

[7] Black, A. P. and Hutchinson, N. "Typechecking Polymorphism in Emerald". Tech. Rep. CRL 91/1 (Revised), DEC Cambridge Research Lab., Cambridge, MA, July 1991.

[8] Gifford, D. K. "Weighted Voting for Replicated Data". *Proc. 7th ACM Symp. on Operating Systems Prin.*, December 1979, pp.150-159.

[9] Jul, E. *Object Mobility in a Distributed Object-Oriented System.* Ph.D. Thesis, University of Washington, Dept. of Computer Science, December 1988. (Tech. Rep. 88-12-06).

[10] Ladin, R., Liskov, B. and Shrira, L. "Lazy Replication: Exploiting the Semantics of Distributed Services". *Proc. of the 9th ACM Symp. on Prin. of Distributed Computing*, Quebec City, Quebec, August 1990, pp.43-57.

[11] Lampson, B. W. "Designing a Global Name Service". *Proc. 5th ACM Symp. on Prin. Distributed Computing*, August 1986, pp.1-10.

[12] Pardyak, P. "Group Communication in an Object-Based Environment". *Proc. International Workshop on Object-Orientation in Operating Systems*, Paris, France, September 1992.

[13] Schroeder, M. S. "Software Clerks". *Proc. ACM SIGOPS Workshop on Models and Paradigms for Distributed System Structuring*, Le Mont Saint-Michel, France, September 1992.

[14] Shapiro, M., Gourhant, Y., Narzul, J. L. and Makpangou, M. "Structuring Distributed Applications as Fragmented Objects". Rapport de Recherche 1404, INRIA, Le Chesnay Cedex, France, January 1991.

[15] van Renesse, R., Birman, K., Cooper, R., Glade, B. and Stephenson, P. "Reliable Multicast between Microkernels". *Proc. of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, Seattle, Washington, April 1992, pp.269-283.

[16] van Renesse, R., Cooper, R., Glade, B. and Stephenson, P. "A RISC Approach to Process Groups". *Proc. ACM SIGOPS Workshop on Models and Paradigms for Distributed System Structuring*, Le Mont Saint-Michel, France, September 1992.