

# Modules, Objects and Distributed Programming: Issues in RPC and Remote Object Invocation

HENRY M. LEVY AND EWAN D. TEMPERO

*Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, U.S.A.*

## SUMMARY

Distributed programming can be greatly simplified by language support for distributed communication, such as that provided by remote procedure call (RPC) or remote object invocation. This paper examines design and implementation issues in these systems, and focuses on the influence of the communication system on a distributed program. To make the discussion concrete, we introduce a single application as implemented in two environments: Modula-2+, an extension of Modula-2 with RPC, and Emerald, an object-based language that supports remote object invocation. We show that small differences in the implementation of the communication system can have a significant impact on how distributed applications are structured.

KEY WORDS Distributed programming Object-oriented programming Programming languages Abstract data types Remote procedure call

## 1. INTRODUCTION

Distributed programming using remote procedure call (RPC) or remote object invocation is now becoming common. In this paper, we examine some of the fundamental design and implementation issues in these communications systems from the point of view of their impact on distributed applications. Our discussion has multiple goals. First, we survey issues in distributed programming using RPC and remote object invocation. Secondly, we present two specific distributed programming systems and show the programming choices motivated by those systems. Thirdly, we show how low-level implementation decisions in an RPC system can substantially affect the design of a distributed program. Fourthly, we draw distinctions between separate concepts that are often used synonymously. For example, we believe that there is a substantial difference between a distributed server and a distributed program: a distributed server is a module that exists at one or more locations on a network and responds to client requests; a distributed program is a single logical task that has been subdivided into co-operating, geographically distributed, parts. It is common for a client/server paradigm to be used as a model for all distributed programs, and this may lead to restrictions that are inappropriate. As another example, we distinguish the concepts of location-transparent naming, in which an

0038-0644/91/010077-14\$07.00  
© 1991 by John Wiley & Sons, Ltd.

*Received 29 November 1989*  
*Revised 31 July 1990*

object can be referenced in a location-independent way, and location-transparent programming, in which location is completely invisible to the programmer.

To make our discussion more concrete, we introduce an example application, called *Doodle*, as implemented in two distributed environments. *Doodle* is a distributed program that manages ‘whiteboards’, which are windows that several workstation users can write or draw on simultaneously, with each seeing an up-to-date image of the whiteboard’s state on his or her screen.

The first implementation of *Doodle* was written in the Modula-2+ language<sup>1</sup> for the DEC Firefly,<sup>2</sup> an experimental multiprocessor workstation built at the DEC Systems Research Center. Modula-2+ is an extension of the Modula-2 language<sup>3</sup> that, together with the Firefly’s operating system, Topaz, includes support for remote procedure call. For comparison, we implemented a subset of *Doodle* in Emerald,<sup>4-6</sup> an object-based language for writing distributed programs. Emerald executes on a small network of MicroVAX or Sun workstations.

Our goal is not to praise or criticize either of these two programming systems, but to use the comparison to expose design issues in distributed communications. We believe that they are good general representatives of the RPC or invocation-oriented approach to distribution. However, one could certainly build an RPC system that operates like Emerald (e.g. Hermes<sup>7</sup>), or an object-oriented one that operates like Modula-2+.

Previous discussion has focused on the similarities and differences between procedure call and message passing in centralized systems.<sup>8</sup> Others have focused on the implementation of RPC and the differences between local and remote procedure call.<sup>9,10</sup> Our objective is simply to use two programming systems to help us examine the influence of different models and implementations on distributed programming. To do this, we examine the *Doodle* application from the perspective of a number of issues: binding, locality, client/server vs. object structure, calling semantics, naming, failure handling and program structure.

Finally, we should emphasize that the issues we discuss cross simple conceptual boundaries; that is, in comparing the two example systems we are sometimes concerned with the programming language, sometimes with the run-time support, and sometimes with the communication model. At various points, it may seem that we are mixing independent factors. In fact, we *are* doing this, precisely because most existing systems have mixed these independent features in a corresponding way.

The following section describes the distributed programming paradigms common to programming in these two languages, and presents model *Doodle* implementations. [Section 3](#) enumerates the issues that we believe are fundamental, and examines the trade-offs in implementing these issues. [Section 4](#) summarizes our discussion.

## 2. PROGRAMMING MODELS

Every programming system, whether explicitly or implicitly, motivates a specific style of programming. As previously stated, one of our goals is to examine the programming styles and design choices motivated by RPC and remote object invocation systems. In this section, we first discuss the client/server model, which has become the basis for many distributed systems. We then examine the programming model used by distributed object-based systems.

**The client/server model**

An important basis of most RPC systems, and a powerful paradigm in distributed computing, is the client/server model. This model has been extremely useful in the design, analysis, and growth of distributed systems. The principal objective of this programming style is to make a collection of (possibly replicated) distributed services available on a network of computers or workstations. A *server* is an executing instance of one of these services, and a *client* is a program that makes use of the service at some time during its execution. In an RPC system, clients communicate with servers using procedure calls. These calls are intercepted by *stub* procedures that are linked with the client and server. Along with the RPC run-time system, these stub procedures facilitate the distributed communication, parameter passing, and so on.

As an example of the use of this model, Figure 1 shows the structure of the Modula-2+ Doodle implementation. The organization is based on the concept of a single Doodle server with many Doodle clients. The structure of the server is shown at the bottom of Figure 1. The server consists of a server program and local data structures. Its main data structure, *ActiveList*, is a list with entries for those whiteboards that are currently in use. Each entry consists of two parts: the whiteboard itself, which includes its name and state (i.e. all the text, lines and curves currently written on the whiteboard), and the list of current users.

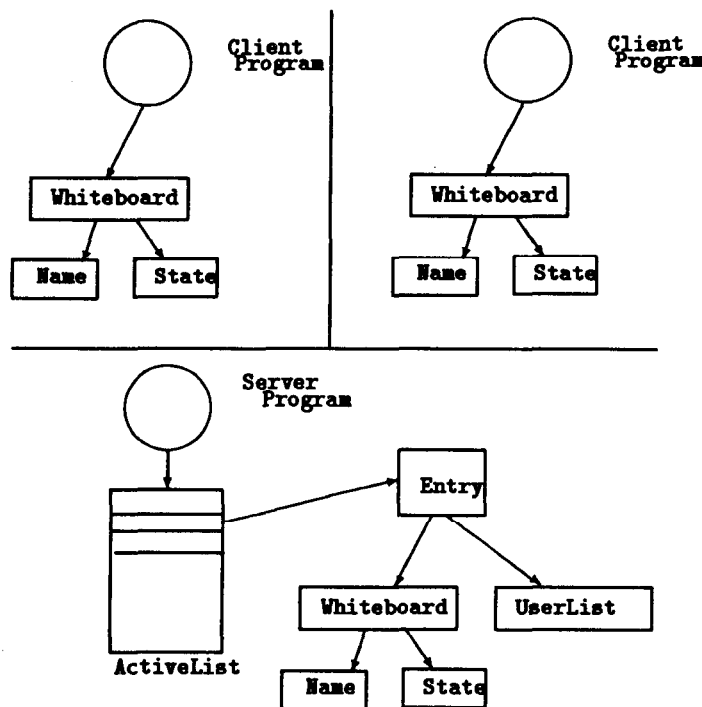


Figure 1. Modula-2+ Doodle implementation

Each Doodle user runs a copy of the client program, which maintains local data structures on his workstation. The client-local data structures are essentially local copies of the server's data structures for the user's currently active whiteboard; these are replicated for performance reasons. The top of [Figure 1](#) shows this structure on two different workstations.

The RPC-based Doodle system operates as follows. A user wishing to run Doodle executes a copy of the Doodle client program, which establishes a connection to the Doodle server (this process, called binding, is described in detail in [Section 3](#)). Once the binding has completed, the user opens a whiteboard of interest by name. If that whiteboard exists, the server will add the user to the whiteboard's userlist. The server then returns a copy of the white board's current state to the client program, which displays an image of the whiteboard on the user's display. Otherwise, the server creates a new whiteboard with the specified name, makes a new entry in its active list (initializing the userlist), and informs the client.

When a user writes to a whiteboard, the client program first modifies the local copy of the whiteboard's state and the image on the user's display. This is mainly a performance issue: it is most important that the drawing user have real-time feedback on his changes. The client then sends the changes to the server, which updates the global data structures and propagates the changes to other users of that whiteboard.

### The object model

Distributed object-oriented systems are a natural outgrowth of object-based operating systems and languages whose use began in the late 1960s and early 1970s. In these systems, every resource or abstraction is represented by an object. An object consists of some private data and the private and public operations (procedures) that manipulate that data. Each object is referenced via a unique name or address.

In a distributed object-based system, objects can reside on any node. Objects communicate by *invoking* operations on other objects. An invocation is essentially a (remote or local) procedure call; the invocation specifies a target object, the name of a public operation on that object, and some parameters. Invocations, like remote procedure calls, are typically synchronous. The binding of the calling object to the called object, as well as the parameter passing, is handled dynamically by the run-time system. In most distributed object-based systems, all invocations (at least conceptually) pass through the kernel or run-time system.

[Figure 2](#) shows the structure of the Emerald implementation of Doodle. In this implementation, all of the data structures are implemented as Emerald *objects*. Furthermore, the system is organized in three somewhat separate parts.

On the left of [Figure 2](#) is a whiteboard directory object. This object maintains a list, similar to `ActiveList` in the Modula-2+ implementation, with entries for every whiteboard in the system. Each entry object, as shown in the centre of the Figure, consists of references to a whiteboard object and its associated userlist object. (These objects could exist in one of several locations, which we will discuss later.) Each whiteboard object has references to its name and state objects.

The right-hand side of [Figure 2](#) shows the structure of the Emerald-based Doodle system on one client node. Each Doodle user runs a copy of the client object, which provides the client interface. The client object has a reference to the entry in the whiteboard directory's list that contains the whiteboard it is using, and a reference to a whiteboard object that is the client's local copy.

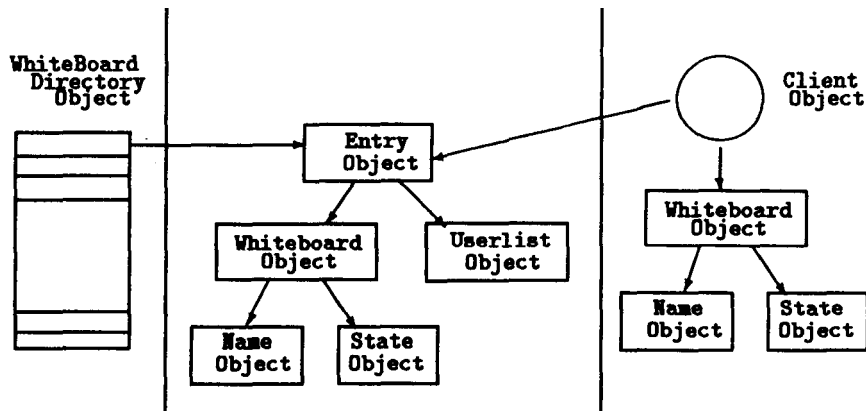


Figure 2. Emerald Doodle implementation

A user opens a whiteboard by specifying the whiteboard's name. The client object sends this name, together with a self reference, to the whiteboard directory. The directory examines its list for an entry with the named whiteboard. If there is an entry with a whiteboard of that name in the list, it will reply with a reference to that entry. Otherwise it will create a new entry for a whiteboard of that name and return a reference to that entry. In either case, the directory adds a reference for the client to the whiteboard's userlist, and returns a whiteboard reference to the client for the client's use.

Whenever a client writes to a whiteboard, it modifies the local whiteboard's state, along with the image on the user's display. The client then modifies the whiteboard object whose reference it received from the directory. It does this by invoking that whiteboard object. That whiteboard object then notifies the other clients of the changes by using references on the userlist to contact them.

### 3. ISSUES

This section presents some of the issues in the design and implementation of RPC-based and invocation-based distributed applications. These issues focus our discussion of the two systems (Modula-2+ and Emerald) and help us to describe trade-offs and to understand how each system operates. Many of these issues are closely interrelated and although the discussions may overlap, different sections take different points of view. The Doodle system, introduced above, is used in some sections to demonstrate the application impact of the various trade-offs.

#### Binding

Binding is the process of establishing communication between two parties. Through binding, the parties determine the addresses of their partners and the protocols to be used for the communication. On a single node, modules communicate using procedure call and return, and address each other using virtual addresses; the binding is performed by the compiler or linker. In a network, there are various choices for how and when binding is made, and these choices affect both the performance and

flexibility of the application. In the most static case, binding is performed at compile time. That is, the two communicating programs know the explicit network locations of their partners. In the most general case, binding can be performed at every interaction. Options exist in the middle that permit trade-offs between flexibility and performance; RPC systems have typically chosen early binding, which permits call-time performance optimizations, whereas invocation-oriented systems have chosen late binding, which permits flexibility.

In most RPC systems, including Modula-2+, binding is performed at run-time, typically when a client program begins execution. The binding is explicit, i.e. it is programmed by both the application and server programmers. An important advantage of this design is that RPC can be implemented as a run-time system without modification of compilers or linkers.

When a server begins executing, it issues an *export* call, which registers the server's name and address with a network-wide name service. When a client begins executing, it issues an *import* call, which checks the name-server database and returns the appropriate addressing information to the client's RPC run-time system. The run-time system on the client node then performs a remote *bind* call to the server node, which establishes the connection to be used for the duration of the client/server interaction.<sup>11</sup> This binding process typically uses a network name-server, such as Grapevine.<sup>12</sup> Once binding is completed, the client can issue remote procedure call requests to the server.

In Emerald, binding is performed by the run-time system (the Emerald kernel) on every call. When one object wishes to interact with another, it invokes an operation on the target object, using a handle (address) that it has obtained, typically through a directory look-up or similar operation. Conceptually, on each invocation, the Emerald kernel must examine the target object address and locate the object. Emerald does not have a centralized name server. Instead, it uses a protocol for querying other kernels in order to locate the target object.<sup>5, 13</sup> In the worst case, this requires a type of reliable broadcast to all kernels. However, once an object is located, the caller's node remembers where that object was last known to reside. An invocation to that object is thus sent to its last known residence; if the object has moved, that node is likely to know where it went.

For static (compile-time) binding, the client and server must each exist at one and only one location. If either node goes down, neither the client nor the server can run. The advantage of RPC-style run-time binding is that the client can locate an appropriate server (possibly one of several) at execution time. A second advantage of explicit binding is call-time performance. Once the binding is completed, however, the client and server must not move. This restriction is not severe, depending on the style of client/server interactions; often in a client/server model the duration of the interaction is short. The programmer could, if needed, explicitly handle server failures and attempt to rebind after the failure. In the case of per-call binding, client and server can move at any time. This mobility can be used, for example, to adjust to changing workloads and to compensate for topological changes. Because per-call binding first assumes that the target has not moved, performance in the normal case is as good as the static binding or RPC-style binding, but flexibility is retained to handle the case where the target has moved, invisibly to the application.

Notice that there are two ways in which binding is visible to the distributed application in RPC systems, and both affect the potential for mobility. First is the

fact that binding is typically performed once at client and server start-up time. Second is the fact that binding must be explicit. That is, even if application-level re-binding is used to handle mobility, only modules that explicitly export themselves would be capable of moving in any case, because it would be impossible to locate them otherwise. If a local module were to move to a remote node, it would need to export itself following the move. Furthermore, clients of the object, some of them formerly local neighbours, would now have to import the module following the move. Finally, the decision of how and when binding occurs seems somewhat orthogonal to the semantics of the procedure call mechanism itself, however it changes the way in which a program must be structured. We discuss this issue more in the next two sections.

### Locality

Selecting the proper location for the components of a distributed application is an important part of its design. Co-locating frequently-communicating elements can improve performance, while replication and decentralization can improve reliability. The correct location of components depends on the requirements of the application and the configuration of the system on which it is running.

In the case of Doodle there are several variations possible. If a whiteboard has only one user then it is reasonable to co-locate the whiteboard and user. In general, this co-location will be more efficient, and with only one user of a whiteboard, there is no advantage to distribution. When there are multiple users of a whiteboard, however, some parts of the state of the whiteboard must be replicated. In this case, either the co-ordination of the whiteboards can be completely distributed among its users, or there can be a centralized whiteboard manager. If the number of clients is large, the whiteboard manager may become computationally expensive, at which point it might be preferable to locate the manager separately from the users. Notice that as the number of whiteboard users changes, parts of the Doodle system may need to move. To allow this flexibility a programming model must provide for the specification of the location of components and have the ability to move them.

In the Modula-2+ implementation of Doodle, the location of the whiteboard server depends on the location of the first client to execute Doodle. (Conceptually it is also possible to start the server independently, however, unless its location is fixed *a priori* by the designer, there is no way to include this location as part of the design.) Whereas the Modula-2+ implementation has one whiteboard server managing all whiteboards, it is possible to split up the management into many servers, for example, to make each whiteboard an independent server. Although this solution can certainly be implemented in Modula-2+, there is in general a fair amount of conceptual, programming, and run-time overhead associated with a server. For example, since each server is typically a separate address space, making each whiteboard a server makes them more expensive to communicate with on the same node. Furthermore, this creates a problem with respect to naming; that is, the name server would have to handle multiple servers with the same name, and the client would have to select the appropriate server instance. Overall, these issues lead programmers to reduce the number of servers in an application, thus causing the creation of fewer but larger servers.

In the Emerald implementation, there is a single whiteboard directory object. The whiteboard directory simply maintains the names and references of open whiteboards. Because every whiteboard is an object, it is inherently a ‘server’, in part because it encapsulates code and data, making it completely self-contained; in comparison, an instance of a data type is a record that contains data only. This fact, in addition to the per-call binding, allows any object to move at any time independently of other objects that it is communicating with. For example, if the creator of a whiteboard closes it while the whiteboard is still in use, the whiteboard object could easily move *itself* to the site of the remaining user; no other objects need to be notified of the move, even though other objects may be in communication with that whiteboard.

In the previous section, we argued that the visibility of binding had an impact on the application and complicated its structure, whereas here we are arguing that location needs to be visible. These may seem to be contradictory points of view, but we believe they are not. Location transparency, often lauded as a goal for distributed programming models, can be examined from two different perspectives. Communications systems should provide location transparency with respect to naming and programming semantics; local and remote communications—messages, procedure calls or invocations—should be identical to simplify programming. However, we believe that distributed systems should not provide transparency to the extent that it is difficult to express location semantics when desired, either to gain location-specific knowledge, or to respond dynamically to that knowledge.

### Clients, servers and objects

Section 2 introduced the client/server model, which has been the basis for many systems. There are, however, a number of restrictions of this model that limit the generality with which distributed applications can be constructed. It is not, in fact, clear whether these restrictions are due to the model itself or to the way in which the model is often implemented. These design choices are not inherent in either the client/server model or in RPC, yet they are common in RPC systems.

For example, an underlying assumption is that servers are well-known, long-lived entities in the system. Because of this assumption, the process of registering a server (the export process) is not considered to be a frequent event and is typically not optimized. In fact, it may take some time before name server updates are propagated throughout the system; during this time, attempts to import a new server may fail. Clients, on the other hand, are assumed to be short-lived entities. Therefore, a client does not register with the name server.

Because of these differences, clients and servers are not equal parties in the communication. First, not all modules are servers because a server module must explicitly export itself, as previously noted. Secondly, the protocol is typically asymmetric and is intended for request/response, master/slave requests: the client sends a query, the server responds, and the communication is complete. In general, the system is not intended either for an exchange of roles (between client and server) or for a relationship between peers. A true peer relationship in an RPC system requires each peer to be both a client and a server.

Suppose we wished the whiteboards within the server’s domain to be servers whereas the whiteboards within the client’s domain are simply local data structures;

this would require two versions of the whiteboard module: a server version and a non-server version. Or, we could always use a server, both locally and remotely, but the local whiteboard server would exist in a separate address space and would still require intra-machine RPCS from the client program to access its operations. These distinctions between conceptually identical implementations add complexity for the application designer.

The non-equality of roles becomes apparent in Doodle because the whiteboard server must asynchronously inform clients of changes to a whiteboard. At that time, the roles have reversed; it is the server who is acting as a client of the client's whiteboard. In many RPC systems, this reversal is not easily made; there is no simple way for the server to call the client. To get around this problem, each client could create additional threads to issue RPC requests to the server. The server leaves these requests unanswered; then, when the server wishes to contact the client, it does so by responding to one of the outstanding RPCS. The existence of such threads causes additional design and implementation headaches in the client; for example, the threads may have to be shut down and restarted if an error occurs.

This problem can be easily solved in RPC systems by including a *callback* facility. Callback permits the server to determine the address of a client and to issue an RPC to that client either during or following the client-to-server RPC. Callback can be implemented by including either a visible or invisible parameter with RPC requests, namely the RPC binding information for the client. The RPC binding information is maintained as an opaque record that contains the low-level client-specific addressing information; it must be maintained in this way because the client has not registered with the name server, so a name cannot be used to specify the target of the callback. Callback greatly simplifies this server-client interaction, but still, in cases where client and server wish to be peers, communication is not equally simple in both directions.

We noted previously that in Emerald, every whiteboard is a server. In fact, in an object-oriented system like Emerald, *everything is* a server because every object can respond to remote (and local) invocations. The only difference between a remote object and a local object is that a reference to the remote object has crossed machine boundaries. This means that the same whiteboard object can be used by the client and server, and the choice of placement for the whiteboard object can change dynamically. Furthermore, the whiteboards can easily invoke each other, since object addresses are easily passed as parameters (in fact, they are the only parameters); the object-to-object relationship is inherently symmetrical.

### Calling semantics and naming

An important issue in distributed systems is the parameter-passing semantics of remote procedure calls or remote invocations: how should parameters be passed across machine boundaries? In non-distributed systems, the programmer typically has a choice of call-by-reference or call-by-value. However, in most distributed systems, call-by-reference is not possible because addresses cannot be passed across the network; therefore, call-by-value (or call-by-value-result) must be used.

Still, with call-by-value, the system must be able to transmit the addressed entity, which may be a data type instance, across the network. The process of placing parameters into a message packet, and later taking them off and reconstructing the

data, is called *marshalling* in an RPC system. Marshalling can be done in several ways, depending on the support provided to the programmer by the RPC system. In some RPC systems, a programmer must explicitly code calls to marshalling procedures, a tedious and error-prone process. In most systems a *stub compiler* program reads a high-level specification of a server's interface and automatically produces calls to marshalling procedures; these calls are part of the stub called locally when an RPC occurs.<sup>11,14</sup> In the Argus system, each abstract data type can be asked to encode itself into a form suitable for transmission to other systems.<sup>15</sup> This gives the programmer the flexibility to define a standardized on-the-wire format for a data type.

The major problem is not that call-by-value and call-by-reference are different, but that call-by-value replicates a data type instance during the duration of the RPC call. Assuming that concurrency exists in a distributed system, each instance represents a state that is potentially shared among several processes. Concurrent access to that state must be controlled through a synchronization mechanism, such as monitors. In fact, a principal objective of abstract data types is the synchronization of shared access, as well as implementation independence.<sup>16</sup> In a call-by-value RPC, however, the server is not accessing shared state, but is accessing a *copy* of the state; at the time the argument is accessed, it may be out of date. Similarly, with call-by-value-result, the effect will be the same as call-by-reference *only* if no other accesses to that state occurred on the source node in the duration of the call. The result is that all shared instances must be controlled by servers, since only the server can guarantee sequential access if needed. Having a data type with concurrency control is not sufficient if an instance of that data type can be passed as a call-by-value-result parameter. This is an issue that exists in centralized systems independent of RPC; however, the problem is greatly magnified by the existence of true concurrency in distributed systems.

An alternative approach to the problem of calling semantics and naming is the distributed virtual memory implemented in the Ivy system.<sup>17, 18</sup> In Ivy, a single address space exists across nodes; the memory manager traps program accesses to shared store and exchanges coherency messages with memory managers on other nodes. From the programmer's point of view, distributed processes communicate through shared memory as they would on a centralized system. Distributed processes can therefore exchange and share addresses. If one built an RPC system on top of Ivy, call-by-reference could be trivially used for parameter passing.

In distributed object-oriented systems, it is also possible to have a single object name space that spans machine boundaries. Each object has a network-wide unique identifier (its ID or address) that can be used to reference the object in a location-independent manner. This obviously implies a level of indirection; all object references must occur through a descriptor mechanism so that remote references can be detected and trapped. Given such a mechanism, call-by-reference can be provided easily. In fact, conceptually the only parameter-passing mechanism in an object-oriented system is call-by-object-reference. Thus, to give someone access to an object, all one does is to pass its address, independent of the location of the object or the receiver of its address.

The network-wide object name space can be managed in several ways. In Emerald, each object has a unique ID. Each node manages its own virtual address space and maintains a table to map IDs into local virtual addresses. As an alternative, a

merging of the Ivy and Emerald scheme occurs in Amber,<sup>19</sup> an Emerald successor, in which a single virtual address space is maintained across nodes, with the virtual address being used as the network-wide unique object identifier. An object's data exists only on one node at a time, and an invocation to a remote object is trapped in software, causing a remote invocation.

The problem with call-by-reference in a distributed object-oriented system is performance. When one object performs a remote invocation of another, passing it references to parameter objects, the receiver will probably cause remote invocations when it attempts to invoke those parameters. In many cases, these remote invocations are unavoidable. However, there are cases in which a call-by-value mechanism would make more sense with respect to performance. For example, if the parameter object is immutable, then sending a copy of it would be semantically equivalent to call-by-reference. Obviously this can be done for immutable objects such as integers, as well as for structures of immutable.

It may also be possible to decrease the cost of a remote call-by-reference parameter by moving the parameter object to the calling site for the duration of the call. The Emerald system has this capability, and moving objects has been shown to have performance improvement potential, depending on the nature of the parameter object being moved (e.g. its size and the number of active invocations). Once again, moving an object is different than call-by-value because the object represents shared state<sup>5</sup>

In some RPC systems, the parameter-passing semantics may differ depending on whether the server is remote or co-resident on the client's node. The difference may be due to performance optimizations using shared memory that become possible in intra-node RPCS. This has the potential to cause subtle errors, particularly with concurrent processes, if a parameter is passed sometimes by value-result and other times by reference.

Once again, the problem for the application is that local and remote communication may be different in some systems. This adds complexity to program design and implementation, and can lead to subtle bugs. For example, a correctly operating client could fail when, following a reconfiguration, a facility that was previously available locally becomes available remotely.

It is somewhat unfair to compare Emerald and Ivy with Modula-2+ with respect to naming, and it is important to point out a crucial difference in goals. Emerald and Ivy provide an integrated distributed programming environment, in particular, with a unified global address space. In contrast, Modula-2+, and RPC systems in general, attempt only to simplify *communication* by unifying local and remote communication through a procedure call paradigm. There is a substantial difference between building an integrated distributed environment and simply integrating remote communication into a programming language. An advantage of the RPC approach, however, is that it can more easily accommodate heterogeneity of language, operating system, and hardware environments.<sup>20</sup> Furthermore, RPC is easily integrated into a traditional programming system.

### Failure handling

A major difference between centralized and distributed systems is in their failure modes. Failures are a difficult matter in distributed systems, and particularly in

systems such as Emerald in which distribution is transparent. In both RPC and distributed object systems, much effort goes into making remote and local calls identical to the extent possible in each particular system. In general, hiding the details of distribution from the programmer simplifies the programming of distributed applications. Failures constitute a notable exception to this rule.

Although it is possible to make local and remote invocations largely identical, it is not possible to make remote invocations obey all of the properties of local invocations. For example, whereas local procedures on a single node can raise error conditions, they do not time-out, they do not become temporarily unreachable and they do not crash independently of the calling procedure. Thus, a local procedure call is typically not equipped to handle such failures.

As we have noted, there is a difference between making distribution transparent from a programming language point of view (i.e. hiding the details of the remote communication) and making the existence of distribution invisible. In an RPC-based system, programmers typically know which calls are to remote servers, and which are not. This explicit knowledge makes it possible to handle remote server failures in a sensible way.

In Emerald, on the other hand, *every* object invocation is potentially remote. As we have seen, this in part gives the system its flexibility, and in the absence of failures, there should be no problem. However, in the presence of failures, it is difficult to imagine coding every invocation as if it might fail due to errors in accessing a remote object. One solution to this might be to add pragmas to the language indicating what invocations or objects are potentially remote.

In either case, the transactional model might be more appropriate for dealing with failures than explicit failure handlers. Transactions have been integrated both with object-oriented and data-oriented distributed systems to provide a more uniform approach to programming in an environment where failures are expected .<sup>21,22</sup>

#### 4. CONCLUSIONS

The objective of this paper was to examine design issues and design decisions in the construction of distributed programming environments and communication systems. To make the discussion concrete, we examined a single application, called Doodle, as implemented in two environments: Modula-2+, an extension of Modula-2 with RPC, and Emerald, an object-based language that supports remote object invocation. We have seen that low-level design decisions in both language support and interprocessor communication can affect the structure of a distributed application, sometimes in subtle ways. Many of these design decisions affect the generality with which distribution can be supported, as well as the ability of the application to adapt to changes and failures at run-time.

Perhaps the most important message is that there is a significant difference between providing support for distributed servers and providing support for distributed programs. A distributed server is a module that resides on one or more network nodes and responds to queries from its clients. Servers tend to be rather static, long-lived entities. On the other hand, a distributed program is a single program whose components may be distributed across several nodes. A distributed program may have more dynamic requirements for using distributed resources.

We believe that the difference between object-oriented and data-oriented languages is more pronounced in a distributed system. In a single sequential system, the encapsulation provided by objects (the integration of code and data) is a logical concept. In a distributed system, the encapsulation is physical; it simplifies mobility and the enforcement of location-independent addressing.

One important issue is the extent to which the programming model encompasses all of the concepts needed to build a program in the intended environment. Most early distributed programming systems simply provided message primitives at the operating system level. The programmer used whatever language was appropriate, and then manually handled communication through run-time calls. The communication was completely outside of the scope of the programming language, and the programmer had to package data explicitly within message packets, and multiplex or demultiplex messages. At this point, a large amount of detail about the communication system was visible and necessary for distributed programming.

Remote procedure call removes much of the detail that was required of programmers in message-oriented systems. Programmers in RPC systems can ignore the complexities of sending and receiving messages, building packets and so on. However, as our discussion has shown, some of the details of the communication system are still visible, although to a much smaller extent than before. Furthermore, some of the distribution facilities visible to the programmer are a result of the programming language, some are part of the RPC programming system, and some are the result of the underlying RPC run-time system.

More integrated systems, such as Emerald or Ivy, completely remove the distinction between the run-time system and the programming language. This can be done in one of two ways, either by bringing the complete distributed programming model into the language, as is done in Emerald, or by implementing a uniform, distributed virtual memory under the programming language, as is done in Ivy. The advantages of this approach have been described in this paper. The disadvantage of this approach, as previously mentioned, is that it requires the distributed program to operate in a separate environment, making it more difficult to integrate into existing systems.

In conclusion, the greater the extent to which different levels of implementation (e.g. language, stubs and run-time system) are visible to the programmer, the greater the complexity of programming. Finally, the differences we have discussed are not inherent in RPC or object-oriented systems, *per se*; they are just dimensions in the design space that could be incorporated into either system style.

#### ACKNOWLEDGEMENTS

We should like to thank the members of DEC Systems Research Center who supported the construction of Doodle, and who participated in an early discussion of this work; in particular, we would like to thank Andrew Birrell and Roy Levin. Luca Cardelli supported the development of Doodle at SRC. At the University of Washington, valuable reviews were provided by Tom Anderson, Kathy Armstrong, Brian Bershad, Jeff Chase, Sabine Habert, Kevin Jeffay, Ed Lazowska, Cliff Neumann and Rajendra Raj.

This work was supported in part by the National Science Foundation under Grants No. CCR-8619663, CCR-8700106, and CCR-8907666, and by the Digital Equipment Corporation Systems Research Center and External Research Program.

## REFERENCES

1. P. Rovner, R. Levin and J. Wick, 'On extending Modula-2 for building large, integrated systems', *Technical Report #3*, Digital Equipment Corporation Systems Research Center, Palo Alto, California, January 1985.
2. C. P. Thacker, L. C. Stewart and E. H. Satterthwaite, 'Firefly: a multiprocessor workstation', *IEEE Trans. Computers*, **C-37**, (8), 909-920 (1988).
3. N. Wirth, *Programming in Modula-2*, Springer-Verlag, New York, 1982.
4. A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, 'Distribution and abstract types in Emerald', *IEEE Trans. Software Engineering*, **SE-13**, (1), 65-76 (1987).
5. E. Jul, H. Levy, N. Hutchinson and A. Black, 'Fine-grained mobility in the Emerald system', *ACM Trans. Computer Systems*, **6**, (1), 109-133 (1988).
6. R. K. Raj, E. D. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson and E. Jul, 'Emerald: a general-purpose programming language', *Software-Practice and Experience*, **21**, 91-118 (1991)—this issue.
7. A. P. Black and Y. Artsy, 'Implementing location independent invocation', *IEEE Trans. Parallel and Distributed Systems*, **1**, (1) 107-119 (1990).
8. H. C. Lauer and R. M. Needham, 'On the duality of operating system structures', *Proceedings of the Second International Symposium on Operating Systems*, October 1978. Reprinted in *Operating Systems Review*, **13**, (2), 3-19 (1979).
9. B. J. Nelson, 'Remote procedure call', *Technical Report CSL-81-9*, Xerox Palo Alto Research Center, May 1981. (Also, *Ph.D. Thesis*, Carnegie-Mellon University, CMU-CS-81-119).
10. G. T. Almes, 'The impact of language and system on remote procedure call design', *Proceedings of the Sixth International Conference on Distributed Computing*, May 1986, pp. 414-421.
11. A. D. Birrell and B. J. Nelson, 'Implementing remote procedure calls', *ACM Trans. Computer Systems*, **2**, (1), 39-59 (1984).
12. A. D. Birrell, R. Levin, R. M. Needham and M. D. Schroeder, 'Grapevine: an exercise in distributed computing', *Communications of the ACM*, **25**, (4), 260-274 (1982).
13. R. J. Fowler, 'Decentralized object finding using forwarding addresses', *Ph.D. Thesis*, University of Washington, December 1985. Department of Computer Science Technical Report 85-12-1.
14. M. B. Jones, R. F. Rashid and M. R. Thompson, 'Matchmaker: an interface specification language for distributed computing', *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, January 1985, pp. 225-235.
15. M. Herlihy and B. Liskov, 'A value transmission method for abstract data types', *ACM Trans. Programming Languages and Systems*, **4**, (4), 527-551 (1982).
16. C. A. R. Hoare, 'Monitors: an operating system structuring concept', *Communications of the ACM*, **17**, (10), 549-557 (1974).
17. K. Li, 'Shared virtual memory on loosely coupled multiprocessors', *Ph.D. Thesis*, Yale University, September 1986. YALEU/DCS/RR-492.
18. K. Li and P. Hudak, 'Memory coherence in shared virtual memory systems', *ACM Trans. Computer Systems*, **7**, (4), 321-359 (1989).
19. J. S. Chase, F. G. Amador, E. D. Lazowska, H. M. Levy and R. J. Littlefield, 'The Amber system: parallel programming on a network of multiprocessors', *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, December 1989, pp. 147-158.
20. B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo and M. Schwartz, 'A remote procedure call facility for interconnecting heterogeneous computer systems', *IEEE Trans. Software Engineering*, **SE-13**, (8), 880-894 (1987).
21. B. H. Liskov and R. W. Scheifler, 'Guardians and actions: linguistic support for robust distributed programs', *ACM Trans. Programming Languages and Systems*, **5**, (3), 381-404 (1983).
22. A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya and P. M. Schwartz, 'Support for distributed transactions in the TABS prototype', *IEEE Trans. Software Engineering*, **SE-11**, (6), 520-530 (1985).