Observations on Building Distributed Languages and Systems

Richard D. Schlichting
Gregory R. Andrews
Norman C. Hutchinson
Ronald A. Olsson†
Larry L. Peterson

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

Abstract: This paper surveys projects in distributed languages and systems at the University of Arizona, and offers observations based on the experience gained during their design, implementation, and use. The relevant projects are the SR distributed programming language, the Saguaro distributed operating system, the MLP system for constructing distributed mixed-language programs, the object-based distributed programming language Emerald, and the Psync interprocess communication mechanism. The observations address the experimentation process itself as well as the design of distributed software.

1. Introduction

Over the past several years, we have been active in the design, implementation, and use of distributed programming languages and systems. These projects have varied greatly in intent and scope, from a full-fledged distributed operating system to a modest system that facilitates distributed, mixed-language programs. Despite their variety, these projects exhibit a surprising number of common elements. While some of these elements are a natural result of the interrelationships among the projects, others have originated from such diverse sources that they represent lessons with broad applicability. As a result, we feel it is possible to draw several conclusions about the process of designing and implementing software for distributed systems.

In this paper, we discuss the experience we have gained in the course of five projects, emphasizing the common threads between the projects and the lessons learned from them.

†Division of Computer Science, University of California, Davis, California 95616

The five projects are the SR distributed programming language, the Saguaro distributed operating system, the MLP system for constructing distributed mixed-language programs, the object-based distributed programming language Emerald, and the Psync interprocess communication mechanism. As alluded to above, there are many interdependencies among these projects. Some of the links are direct; for example, MLP is an outgrowth of Saguaro and has also been heavily influenced by SR. Other links are more indirect; for example, although Psync was influenced by the desire to provide interprocess communication support for stand-alone versions of SR and Emerald, its design evolved independently from the rest of the projects. In other words, each project has been influenced to a certain degree by the others. Indeed, as discussed in Section 3.3 below, we feel that one of the most important advantages of having multiple ongoing projects in the same general area is that they can draw on each other as a source of inspiration and ideas.

This paper is organized as follows. Section 2 summarizes the five relevant projects, roughly in chronological order by date of inception. Section 3 then elaborates on the lessons we have learned from these experiences and offers advice to others engaged in similar activity; these observations address not only the actual design of such software, but also the experimentation process itself. Finally, Section 4 contains some conclusions.

2. Overview of Projects

2.1. The SR Language

During the past two years we have redesigned and reimplemented the SR (Synchronizing Resources) programming language. Like its predecessor, SR_0, SR remains a language for writing distributed programs. Also, the main language constructs—resources and operations—are conceptually the same. However, based on our experience using SR_0 to write numerous programs, including prototypes of the Saguaro operating system, we have modified the language in several ways. In essence, SR is to SR_0 what Modula-2 is to Modula: a second-generation language that incorporates refinements based on experience with its predecessor.

The redesign of SR has been guided by three major concerns: expressiveness, ease of use, and efficiency. By expressiveness we mean that it should be possible to solve distributed programming problems in the most straightforward possible way. This argues for having a flexible set of language mechanisms, both for writing individual modules and for combining modules to form a program. Distributed programs are generally much more complex than sequential programs. Sequential programs usually have a hierarchical structure; distributed programs often have a web-like structure in which components interact more as equals than as master and slave. Sequential programs usually contain a fixed number of components

since they execute on a fixed hardware configuration; distributed programs often need to grow and shrink dynamically in response to changing levels of user activity and changing hardware configurations. Sequential programs have a single thread of control; distributed programs have multiple threads of control. Thus, a distributed programming language necessarily contains more mechanisms than a sequential programming language.

One way to make a language expressive is to provide a plethora of distinct mechanisms. However, this conflicts with our second concern, ease of use. The way we have resolved this tension between expressiveness and ease of use is that SR provides a variety of mechanisms, but they are based on only a few underlying concepts. Moreover, these concepts are generalizations of those that have been found useful in sequential programming, and they are integrated with the sequential components of SR so that similar things are expressed in similar ways. The main components of SR programs are parameterized resources, which generalize modules such as those in Modula-2. Resources interact by means of operations, which generalize procedures. Operations are invoked by means of synchronous **call** or asynchronous **send.** Operations are implemented by procedure-like **proc** declarations or by **in** statements. In different combinations, these mechanisms support local and remote procedure call, dynamic process creation, rendezvous, message passing, and semaphores—*all* of which we have found to be useful. The concurrent and sequential components of SR are integrated in numerous additional ways in an effort to make the language easy to learn and understand and hence easy to use.

A further consequence of basing SR on a small number of underlying concepts is good performance. SR provides a greater variety of communication and synchronization mechanisms than any other language, yet each is as efficient as its counterpart in other languages. We have also designed the language and implemented the compiler and run-time support in concert, revising the language when a construct was found to have an implementation cost that outweighed its utility. In addition, some of the expressiveness within the language has been realized by "opening up" the implementation. For example, the various mechanisms for invoking and servicing operations are all variations on ways to enqueue and dequeue messages.

An initial implementation of a large subset of SR became operational under Berkeley UNIX[1] on Vaxes in November 1985. Since then, the implementation has been ported to UNIX on Sun workstations and extended to implement the full language except for failure handlers and a few minor features. The current implementation also includes facilities to invoke C functions as operations, thereby gaining access to underlying UNIX system calls. We expect

[1]UNIX is a trademark of AT&T Bell Laboratories.

that the UNIX versions of the implementation will be completed by the end of 1987, at which time they will be made available to interested groups. Work is also underway on a version of the implementation that will allow SR programs to run stand-alone.

Our implementation has been used in graduate classes in concurrent programming, where students have written moderate-sized distributed programs including card games, automatic teller machines, simple airline reservation systems, and prototypes of components of distributed operating systems. Recently, SR has been used to experiment with different upcall and downcall structures [Atki87a], to program a highly parallel interpreter for Prolog, and to program Saguaro's file system.

The new version of SR is defined in [Andr85]. An overview of the language and its implementation is given in [Andr87b]. A detailed discussion of how SR has evolved—what has changed and why, as well as what has not changed and why not—appears in [Andr86]. Much of this work was performed as part of a student's dissertation research [Olss86]. The application of SR to the implementation of the Saguaro file system is described in a second dissertation [Purd87a]. The performance of the implementation is discussed in detail in [Atki87b].

2.2. The Saguaro Distributed Operating System

Over the past four years we have also been involved in the design and implementation of Saguaro, an operating system for computers connected by a local-area network. Systems constructed on such an architecture have several potential advantages due to the multiple processor makeup of the architecture. One of these potential advantages is increased throughput as a result of concurrency. In Saguaro, this advantage is made available to the user through *channels*, an interprocess communication and synchronization facility that generalizes UNIX pipes. Specifically, channels provide the communication and synchronization mechanism to connect the input and output streams of different commands. Each channel has one write port and one or more read ports. Data written to a channel's write port is buffered on each of the channel's read ports. More than one process can write to a channel's write port, in which case the different streams are merged in the order in which the writes are serviced by the channel. Also, more than one process can read from the same read port, in which case each process consumes some subset of the data buffered on that port.

The advantage of channels is that they allow different commands to be connected to form general graphs of communicating processes. Although the power of this facility is largely unexplored, we are convinced based on our experience with constructing distributed programs that certain problems are more easily solved by connecting commands in ways other than the pipeline structure provided by systems such as UNIX. For example, a channel

with N read ports can be used to implement a communication facility in which each process in a group of N processes sees all the messages sent by any member of the group (this is similar to a multicast group [Cher85]). This is accomplished by having all N processes share the channel's write port and having each process read from a different one of the channel's read ports.

Another advantage of a network computer is the potential for increased file availability due to the inherent redundancy of such an architecture. In Saguaro, semi-transparent file replication and access is supported by two mechanisms: *reproduction sets* and *metafiles*. A reproduction set is a collection of two or more files that the system attempts to keep identical. Once a reproduction set is established, modifications to any member of the set are propagated to the other members when the modified file is closed. If a member is inaccessible when the propagation takes place—for example, if the node on which the file resides has failed—an error message is returned to the user. A reproduction set is intended to provide inexpensive user-level file replication for the many applications that do not require guaranteed consistency of a large number of copies in all possible situations. For example, reproduction sets would be a good choice for maintaining multiple copies of a file containing the sections of a paper, but not for maintaining a bank database.

A metafile is a special file that contains symbolic pathnames of other files. When a metafile is encountered during the pathname traversal performed upon file open, one of the names contained in the metafile is selected and used in its place. If the result of the selection is the name of a file that is inaccessible, another name is selected and an attempt is made to access that file. The open fails only when it has been determined that every component file is inaccessible. Thus, a metafile can be viewed as a generalization of the symbolic link facility found in Berkeley UNIX to allow for multiple files and to account for file unavailability. The most common use of a metafile is to provide a single name for the collection of files that comprise a reproduction set.

The Saguaro file system has other interesting features in addition to reproduction sets and metafiles. The logical file systems forms a single tree, yet any file can be placed at the user's discretion in any of the physical file systems. This organization allows, for example, the files comprising a reproduction set to be placed on different disks to enhance availability while still being located in the same directory in the logical file system hierarchy. However, it also means that the normal algorithm for locating a file by traversing the directories in the file's pathname may not succeed in the event of system failure even if the file itself is accessible. This problem is solved in the Saguaro file system by storing additional information about the contents of individual physical file systems in files known as *virtual roots*, and by using a *broken path algorithm* to bypass unreachable intermediate directories.

A final novel aspect of Saguaro is that it makes extensive use of the *Universal Type System* (UTS), a type system containing a type expression language and an external data representation. The type expression language is used in the system to describe user data such as files and to specify the types of arguments to commands and procedures; the external data representation is used as the basis for representing the data stored in system constructs such as files and channels. These uses of UTS enable the system to assist in type checking and leads to a user interface in which command-specific templates are available to facilitate command invocation.

Several papers describe various aspects of Saguaro. The design of the full system is presented in [Andr87a], while the file system mechanisms for supporting high availability are described in [Schl86]. More specific details on the file system and its implementation can be found in [Purd87a]. A related report describing the implementation and use of reproduction sets and metafiles in a UNIX environment has also been written [Purd87b].

2.3. The Mixed-Language Programming System

The Mixed-Language Programming (MLP) System is a simple system for constructing distributed, mixed-language programs that is based on UTS, the type system used in Saguaro. In effect, MLP provides two complementary facilities: the ability to write each procedure of a sequential program in a different programming language and a simple remote procedure call (RPC) facility. These two features taken together allow the user to exploit both language and machine heterogeneity.

From the outset, the primary goals of the system were first, to construct a system that would be useful to the typical programmer, and second, to do so without incurring heavy implementation cost. This second goal was achieved mainly by adopting a philosophy that MLP should not attempt to provide a 100% solution. Rather than design a complex system with the functionality to handle all possible situations that arise in mixed-language programming, we opted instead to design a simple system that can handle the most common situations well. For example, we used separate address spaces rather than attempt to merge multiple languages into a single address space, meaning that arguments can only be passed with value/result semantics. Nevertheless, our experience has shown that the 90% solution achieved by MLP provides a very useful level of functionality at a cost much less than would be required for a fully general system.

A program utilizing MLP is written and executed in several steps. First, one or more *components* are written, each containing procedures or functions written in the same host language. In addition to the procedures, program components also contain interface specifications written using UTS type expressions that describe the number and type of

parameters for each exported or imported procedure. After each component is written, it is translated into object form using the MLP translator for the host language. In addition to performing normal translation, the MLP translator processes the interface specifications and generates code to interface the component with the MLP run-time system. Following the translation of all components, the MLP linker is invoked. This command performs two functions: binding exported procedures names to components and checking types of corresponding import and export specifications. The result of executing the linker is an executable version of the program.

When an MLP program is executed, the component containing the main program is started immediately. Processes for each of the other components are created on the first call to any procedure contained in that component. Data is transmitted between components on the same or different machines using the external data representation defined as part of UTS. During cross-component procedure calls, the MLP run-time systems marshalls and unmarshalls arguments by translating values between their host language format and the corresponding UTS representation as required.

In addition to handling straightforward cross-language calls with a minimum of effort, MLP also contains facilities that allow explicit user control over such functions as data conversion, thereby providing flexibility in more complex situations. In particular, the system provides facilities for handling parameters of UTS types not supported by the host language, *underspecified parameters* whose actual argument can vary in type from call to call, and procedures passed as arguments. The basis for the advanced capabilities of MLP are *UTS representatives*: capabilities or "tickets" for a UTS value that is maintained by the MLP system instead of being automatically translated into a host language value. Each MLP language has been extended with the type **representative**, which is used to declare parameters that are not to be unmarshalled automatically.

To manipulate instances of type **representative**, a collection of operations are available in the form of UTS library procedures. There are three kinds of operations: query operations, marshalling operations, and unmarshalling operations. The query operations allow information about the UTS values associated with a representative to be ascertained, while the marshalling operations provide the ability to translate host language values into UTS format, assign UTS values to representatives, and compose UTS values. The unmarshalling operations provide analogous facilities for converting UTS values into host language values.

Our implementation of MLP has been in use since November 1986 on an interconnected collection of Vaxes and Suns running Berkeley UNIX. Since that time it has been used to construct a mail system, a small database system, and a collection of network transparent plot routines. The system currently supports the programming languages C,

Pascal, and Icon [Gris83].

MLP is described in [Haye87a], while an outline of its implementation and the experience we have gained in using the system appears in [Haye87b]. A user's manual on the MLP system has also been written [Manw86a], as has a report describing how to add new languages to the system [Manw86b]. A complete description of UTS and its application to both MLP and Saguaro will appear in [Haye88].

2.4. The Emerald Language

Emerald is an object-based language and system designed for the construction of distributed applications. The principle feature of Emerald is a uniform object model appropriate for programming both private local objects and shared remote objects. Emerald objects are fully mobile and can move from node to node within a network, even during an invocation. Despite this highly mobile nature of objects, invocation of an operation on an object is location independent; the programmer need not know the location of an object when invoking it. Emerald also supports an abstract type system that concentrates on the specification, not the implementation of objects. Also, note that while both SR and Emerald are distributed programming languages, their intended areas of use are quite different. SR is a systems programming language while Emerald is an applications programming language. As a result, the abstractions supported by the two languages are quite different.

Emerald's goal is to simplify distributed programming through language support, while also providing acceptable performance and flexibility, both locally and in a distributed environment. Like Eden [Alme85, Blac85], Emerald's model of computation is the object. Objects are an excellent way to structure a distributed system because they encapsulate the concepts of process, procedure, data, and location. In contrast to a number of existing distributed programming languages and systems that support separate computational models for local and distributed entities, Emerald supports a single object model. All Emerald entities ranging from Booleans and integers to compilers and entire file systems are programmed using the same model, and have identical invocation semantics.

While we believe that programmers deserve the semantic consistency offered by a single object model, we do not accept the common criticism of object-based systems, namely, that they are too slow. To a limited extent, the Emerald compiler is capable of analyzing the needs of each object and generating an appropriate implementation. For example, an array object whose use is entirely local to another object may be implemented using shared memory and direct pointers, while another array that is shared globally requires a more general (and expensive) implementation that supports remote access.

One novel aspect of Emerald's uniform object model is its support for fine-grained mobility. Mobility in the Emerald system differs from existing process migration schemes in two important respects. First, Emerald is object-based and the unit of distribution and mobility is the object. While some Emerald objects contain processes, others contain only data: arrays, records, and single integers are all objects. Thus, the unit of mobility can be much smaller than in process migration systems. Object mobility in Emerald therefore subsumes both process migration and data transfer. Second, Emerald has language support for mobility. The language explicitly recognizes the notions of location and mobility and provides primitives to discover the locations of objects and move them about in the network. In addition, concern about the efficiency of mobility prompted the introduction of a parameter passing mode, *call-by-move*. Call-by-move combines the movement of an argument object with an invocation, often in a single network packet.

The Emerald language supports the concept of abstract type. The abstract type of an object defines its interface: the number and names of operations that it exports, and the number and abstract types of the parameters to each operation. For example, the abstract type *Directory* specifies that directories implement the operations Add, Lookup, and Delete. Further, Add requires a string and an object (of arbitrary type), Lookup takes a string and returns an object, and Delete requires just a string. We say that an object conforms to an abstract type if it implements at least the operations of that abstract type, and if the abstract types of the parameters conform in the proper way.

Since abstract types capture only the specifications of objects (and not their implementations), they permit new implementations of an object to be added to an executing system. To use a new object in place of another, the abstract type of the new object must conform to the required abstract type. Note that each object can implement a number of different abstract types, and an abstract type can be implemented by a number of different objects.

Emerald has been implemented under Berkeley UNIX on Vaxes and Suns, and is currently running on small networks at the Universities of Arizona, Copenhagen, and Washington. A small number of applications have been implemented, including a mail system, a shared calendar system, a file system, and a replicated name server. In addition, a number of load-sharing style applications have been implemented to experiment with light-weight mobility.

An overview of the Emerald language is given in [Blac86]. The rationale for the design and a description of the compiler algorithms used to deduce appropriate implementations are in [Hutc87] . The type system is described in [Blac87]. An overview of the object migration facility is in [Jul87], and the details of the implementation of the run-

time system including garbage collection will appear in [Jul88].

2.5. Psync

Motivated by the diverse communication needs of distributed applications, including SR, MLP, and Emerald, we have designed and implemented a new interprocess communication (IPC) mechanism called Psync [Pete87]. The communication abstraction supported by Psync, called a *conversation*, provides a "shared message space" through which processes send and receive messages. The novel aspect of the conversation abstraction is that it adds a second dimension to group communication by preserving the *happened before* partial ordering of messages exchanged among the participants [Lamp78]. Just as physical clock signals are encoded with data bits in a raw communication channel to help keep the source and destination synchronized, timing information drawn from a distributed computation's *logical clock* is embedded in the conversation abstraction.

A conversation begins when a process sends an initial message to a set of processes. Once established, a process sends a message in the *context* of all messages that it has received. A *context graph* preserves the context relation and defines the structure of the conversation. Figure 1 shows the context graph for a conversation in which message m1 was the initial message of the conversation; messages m2 and m3 were sent by processes that had received message m1 and message m4 was sent by a process that had received messages m1 and m3, but not message m2.

Psync implements the conversation abstraction by replicating a copy of the context graph on multiple hosts, and forwarding messages between the copies over an unreliable communications network that may lose, duplicate, and deliver messages out of order. The protocol for keeping the copies of the context graph consistent is optimistic in that it sends the

---

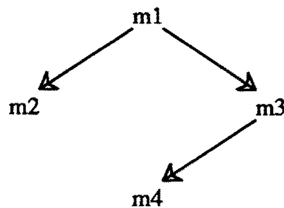

Figure 1 — Sample Context Graph

context in which a given message was sent (i.e., the set of messages that precede the message) only when that context is missing.

A prototype of Psync has been implemented on UNIX where the ARPANET datagram protocol (UDP) provides the underlying communication support. Our experience with the prototype suggests that Psync has two important strengths. First, the conversation abstraction offers a simple and elegant solution to the communication needs of a broad spectrum of distributed applications. For example, we have implemented a remote procedure call mechanism that sends a reply message in the context of the request message; a virtual circuit mechanism that enforces a restriction that the context graph's breadth is bounded by the size of the circuit's sliding window; and an ordered broadcast mechanism that applies an incremental topological sort to the context graph. We believe the elegance of these implementations is due to the fundamental nature of the message context relation in interprocess communication.

Second, an application does not pay a significant performance penalty for using Psync instead of building directly on an unreliable communications network. For example, a client and a server process running on a pair of Sun 3/75s connected by a 10 MB Ethernet can exchange 10 byte request and reply messages in 7 msec using either Psync or UDP. Furthermore, Psync's performance remains constant for large context graphs (i.e., hundreds of messages). This efficiency can be attributed to the implementation strategy: messages are sent asynchronously; there are no connection establishment, termination, or acknowledgement messages exchanged; and the context information enclosed with each message consists of message identifiers (not the messages themselves) and is bounded by the number of participating processes.

Encouraged by this experience, we are currently testing a second implementation in which Psync is embedded in the UNIX kernel adjacent to UDP. The kernel implementation affords the obvious performance improvements: co-local processes share a single copy of the context graph, messages are forwarded to each host that supports a participating process rather than to each process, and the extra message copies needed to add and remove headers are avoided. Work is also underway to use Psync in the SR run-time support system and to implement replicated objects in Emerald.

3. Lessons

In this section, we elaborate on what we feel are the most important lessons learned from our involvement with these projects. These four lessons can be summarized briefly as follows. First, the key in any language or system design is choosing abstractions that strike the right balance between expressiveness and efficiency. Second, the process of designing

and implementing these kinds of systems is inherently difficult and time-consuming. Third, interesting research often has beneficial and unforeseen side-effects. Finally, types are an important and valuable aspect of both language and system design. We elaborate on each of these observations in turn.

3.1. "The right stuff"

The key to any system design effort is developing the right abstractions. The major difficulty in finding the right abstraction revolves around the tension between expressiveness and efficiency. An abstraction needs to be sufficiently expressive to solve the relevant problems, yet implemented efficiently enough to make the system usable. The art of striking the balance between expressiveness and efficiency is strongly influenced by two factors. First, abstractions are dependent on the particular domain of problems for which the system is intended to be used. Designing an abstraction, implementing it, and using it must go hand-in-hand; you have to be willing to rethink and possibly discard early designs based on the experience gained in implementing and using them. Second, abstractions do not exist in a vacuum; a given system supports a blend of abstractions. It is important to consider how abstractions fit together and interact in the system or language.

It is clear from our experience that any given abstraction should be as expressive and flexible as possible. Furthermore, the rules governing a given abstraction should apply uniformly to all its uses, with few or no exceptions. In this way, a user can confidently predict the way a given abstraction is used and what it does regardless of the context. One aspect of the evolution of SR reflects this need for flexibility. The earliest version of SR provided static resources and processes. A subsequent version provided dynamic processes, but static resources. A more recent version provided dynamic resources and processes, but a fixed number of virtual machines. All these objects are now dynamic in the current SR. An interesting point is that although resources and processes were static in the original version of SR, the communications paths between processes could vary during program execution through the use of capability variables. This has proved to be a good decision: the concept of a capability has scaled up in a natural way and now includes capabilities for resources and virtual machines.

SR's communications primitives are another instance where expressiveness has been maximized without sacrificing performance. They provide asynchronous message passing, remote procedure call, rendezvous, and dynamic process creation. We have found all of these useful to solve different kinds of problems; for example, remote procedure call is appropriate for implementing client/server or hierarchical type interactions, while asynchronous message-passing is more useful in situations requiring interacting servers or grid-like

computations. Thus, when taken as a whole, the communication primitives in SR provide a coherent collection that can be used to solve almost any problem easily.

The Emerald object model and type system are also expressive. Because local and distributed objects are defined in the same model, one can use an object that was originally intended to be local in a distributed fashion without any modification to the code that defines it. In addition, the type system allows the choice of an object implementing a particular abstraction to be delayed arbitrarily; the choice can even be changed while the system is executing.

The counterbalancing goal in designing the right abstraction is developing an efficient implementation. Our experience provides several helpful insights regarding efficiency. The most important way to promote efficiency is not to include extra power in the abstraction that is not needed in the target application domain. Consider Psync for example. Although it could be implemented on top of a transaction management system in an effort to ensure that all copies of the context graph remain identical, a weaker — and consequently cheaper — form of consistency is sufficient for the target application domain, i.e., other communication protocols. Specifically, Psync does not guarantee that all messages sent to a conversation are delivered to all copies of the context graph; it only preserves the context relation for those messages that are delivered. Higher level protocols and other applications that need stronger forms of synchronization are easily implemented in terms of the context relation.

Another observation based on our experience is that good abstractions can often be implemented efficiently by optimizing the implementation for the common cases that do not use all of the possible variations provided by a mechanism. In other words, the common cases can and should be implemented efficiently, while the execution costs necessary for the most complicated cases should be paid only by those that use them. The implementation of the input statement in SR is a good example of this approach. In its most general form, a single input statement can service one of several operations and can use synchronization and scheduling expressions to select the invocation it wants. Moreover, an operation can be serviced by input statements in more than one process, which then compete to service invocations. This flexibility is very useful, but most input statements do not require such full generality. Our SR implementation, therefore, provides efficient implementation of the simpler cases; in fact, some operation invocations and input statements are implemented directly as semaphores.

The implementation of Emerald has also demonstrated the value of optimizing for common cases. Emerald objects are mobile, so they can in general move at arbitrary times. This implies that the general instruction sequence for invoking an object must be able to handle correctly objects that are on other machines or are in transit. However, a large fraction

of the objects used in typical applications are private to some larger containing object. Since the compiler has access to all references to such objects, it can detect the common situation where these objects are not moved. In these cases it can generate an improved invocation sequence that takes advantage of the guaranteed locality of the target object. While the general invocation sequence may involve sending a network message to the target object, the invocation sequence for local objects is comparable in cost to a procedure call in a traditional sequential programming language.

A final observation is that efficiency can often be increased and implementation cost cut by scaling back the expressiveness of the abstraction somewhat and implementing a "90% solution". That is, rather than implement a complex system that can handle all possible cases, implement a system that can handle the common cases well. There are many situations where this approach yields a very useful system at low cost and for which it would very expensive to handle the hard cases that comprise the "last" 10%. For example, MLP is a simple system that does not solve all of the many problems associated with mixed-language programming, but rather attempts to provide a useful level of functionality at low cost. The reproduction set and metafile mechanisms in Saguaro also fit into this category: they have been designed specifically for replicating a small number of files in situations that do not require a high degree of consistency in order to avoid the high cost of guaranteeing absolute consistency in all cases.

To summarize, our experience suggests that the best abstractions are those that are the right compromise between expressiveness and efficiency, and that it is worth spending time to develop good abstractions. Moreover, while good abstractions can often be implemented efficiently, it is also worth recognizing the validity of the 90% approach as a way to increase efficiency and lower implementation cost.

3.2. "You want to graduate when?"

Conducting research in language and system design is incomplete without at least a prototype implementation. However, the goal of such an effort in a university environment is not, or at least *should* not be, to produce production quality code for end users. Rather, the implementation part of a research project should be used for evaluating and providing feedback on the decisions that were made during the design process. This allows the strengths and weaknesses of the chosen abstractions to be weighed so that the accumulated experience can affect future versions of the software in a positive way. In the systems that we have built, this kind of feedback cycle has been valuable in designing successive versions of MLP, the file availability mechanisms in Saguaro, and especially SR. For example, based on user feedback, the collection of standard sequential control constructs in SR has been

enriched considerably over the original version of the language, which was rather spartan in that regard (e.g., no indexed do-loop). Other successful research projects such as the Icon programming language [Gris83] and the Cornell Program Synthesizer [Teit81] have also demonstrated the value of this approach.

An implementation is also necessary in order to measure performance. In addition to providing feedback on the particular implementation strategies used, conducting such experiments helps ascertain the inherent costs of the mechanisms in the system. For example, it would be impossible without an implementation to determine the precise relative cost of using a procedure call to invoke a local operation in SR versus an asynchronous send; information on such ''intrinsic costs'' can be quite valuable to a user faced with a decision on what mechanism should be used in a given situation. Moreover, the only way to obtain realistic estimates of these costs is to build the entire system in a manner consistent with its intended end usage. For example, Psync should be built into an operating system kernel since it is an interprocess communication mechanism, while SR should be implemented on top of only minimal system facilities since it is intended to be a systems programming language.

All of these points argue for building an implementation and for making it as realistic as possible. However, we have found—as have many others—that it is almost impossible to overestimate the difficult and time-consuming nature of building such an implementation in an academic setting. One reason, as noted above, is the importance of the feedback cycle in implementation work. This makes the ideal end result not just a single version of the design and its implementation, but rather a series of designs and implementations. This obviously increases the time and resources that must be devoted to a project.

Another reason that research involving implementation is difficult is that many of the tasks that must be done in order to get an implementation up and running are not really research. This is especially true for projects that should really run stand-alone on the hardware. For example, an operating system such as Saguaro requires device drivers and memory management even though these pieces of code are not part of the research aspect of the project. The available options are usually unappealing: excising relevant pieces of existing systems and incorporating them in the implementation, using non-research personnel (e.g., staff) to perform the implementation, or compromising the research nature of the project by having graduate students do the implementation. Another option—building a 90% solution or constructing the system on top of a research vehicle such as Mach or UNIX—compromises the integrity of the performance measurements, but can, as discussed in Section 3.2, still result in a useful system.

One way in which the implementation task can be simplified somewhat is by using tools that automatically generate pieces of the system. Such software is most commonly used

in the programming language area, where tools such as *lex*, *yacc*, and the Amsterdam Compiler Kit (ACK) are often used to simplify the tasks of lexical analysis, parsing, and code generation, respectively. Unfortunately, there are far fewer tools available that are appropriate for building stand-alone systems. In fact, our experiences have convinced us of the need for such tools to the extent that we have recently started the design of a tool called the *x-kernel* that is intended to simplify stand-alone system design. Specifically, the *x*-kernel is a highly configurable communication kernel in which the fundamental unit of composition is the protocol. The *x*-kernel provides the infrastructure—including a uniform protocol interface, an address translation tool, a buffer management mechanism (for attaching headers and trailers to messages), and so on—needed to build a customized protocol or configure a customized communication system. Note also that the building of tools such as the *x*-kernel constitutes research in its own right as well as aiding other research.

While tools are undoubtedly beneficial, they should not be viewed as a panacea. For example, the use of tools can restrict the portability of the resulting software. These problems are most often due to technical problems such as software or hardware incompatibilities. However, other considerations can also affect portability; for example, legal restrictions associated with tools may prevent the resulting software from being distributed to other sites that lack an appropriate license. Another problem with tools is that they can affect the efficiency of the resulting software. This is especially true of systems intended for stand-alone execution; if use of a tool affects the intrinsic costs of the mechanisms to be measured in such a way that it cannot be factored out, then much of the intended benefit of constructing a stand-alone implementation has been lost.

The experience we have gained in implementing the projects described in Section 2 confirms that there are no easy answers to the problems associated with experimental system building. However, we do offer two pieces of advice to others who find themselves engaged in similar activity. First, retain perspective on the goals of the implementation effort; in our opinion, constructing an implementation in a university setting is a research activity that serves to produce a "proof of concept" rather than a polished piece of production code. Second, use tools to simplify the process, or, even better, strive to construct better tools; in our opinion, the biggest need is for configurable tools with the flexibility to be useful in many different situations.

3.3. "Getting there is (at least) half the fun"

One of the biggest lessons we have learned from our work is that the design and implementation process itself often has beneficial and unforeseen side-effects unrelated to the goal of producing a final and complete system. Often, these side-effects take the form of

positive influence on other related projects. For example, the design of the UTS type system and the component structure of MLP were influenced by SR. One project can also influence another by providing a realistic application, thus mitigating the problems that arise when a system is only tested with "toy" applications. For example, Saguaro has served this purpose for SR, while SR, MLP, and Emerald are all well-suited as applications for Psync. These cases also demonstrate the advantages of having a critical mass of related projects, which provides interaction and cross-fertilization of ideas that would not occur if each project operated in a vacuum.

Occasionally, a project does not just influence another project, but actually inspires the creation of a new project. There are at least four instances of such "spinoffs" in our own work. One is MLP, which is based on the type system designed originally for Saguaro. Another is the implementation of reproduction sets and metafiles on top of UNIX, which was inspired by Saguaro. A third spinoff is Psync, whose conversation-based approach to interprocess communication can be traced directly to Dragonmail [Come86]; in essence, Psync provides a conversation abstraction for processes in much the same way that Dragonmail provides a conversation abstraction for users. Finally, Emerald has its roots in Eden; where Eden provides operating system support for distributed object-based, applications, Emerald supports a similar style of programming through a new programming language.

Each of the above examples again demonstrates the advantage of viewing system design and implementation as research rather than production. By taking a broad perspective on the process rather than focusing on the narrow target of producing a piece of software, a project can often act as a catalyst for other projects. This philosophy also implies that if a project is interesting and worthwhile, benefits are likely even if the project is never completed in the originally envisioned form or if the project happens to change direction.

3.4. "Types are a system's best friend"

A final lesson concerns the importance of types in many different settings. One major role of types in languages and systems is for specifying interfaces, i.e., the number and types of arguments. In SR, a signature is used to specify the interface of an operation, while UTS expressions are used in Saguaro and MLP to describe the interface of commands and procedures, respectively. The Emerald type system takes this to the extreme; in fact, it is used *only* to specify interfaces, with no capability for specifying implementations.

One conclusion we have been able to draw from our experience with SR, Saguaro, MLP, and Emerald is that it is important to provide a flexible type system, especially if the system supports dynamic communication. In SR for instance, communication paths are

established dynamically by using capabilities so structural equivalence is used for type checking; this approach is much less restrictive than the name equivalence test used by languages such as Ada. As another example, consider the underspecified parameter facility of UTS, which allows an interface to be specified so that the type of an argument can vary from call to call; this facility can be used to write polymorphic procedures or commands and helps support the dynamic command connections found in Saguaro.

Another conclusion from our work is that types can be valuable in systems as well as programming languages. Specifically, we have found at least four advantages to our use of types in Saguaro. First, it enables the operating system to guarantee that the arguments to a command are of a type it expects; this simplifies the coding of commands by reducing the need to parse argument lists or interpret data read from files or channels. Second, since the interface specification associated with each command specifies completely how that command is to be invoked, command-specific templates containing information about a command's arguments can be used in the user interface to facilitate the construction of an invocation. Third, the use of types provides enough flexibility that the distinction between commands and procedures can be blurred to the point that commands can be invoked as procedures and vice-versa. Finally, the use of a type scheme like UTS that specifies an external data representation makes it easier to accommodate heterogeneity in languages and architectures by providing, in essence, a common communication language; this common language is, of course, the basis for the MLP system.

Our experience with types in both distributed languages and systems can be summarized as follows. First, type systems are useful for specifying interfaces, but it is important to provide enough flexibility to facilitate dynamic communication. Second, it is worth considering using types even in contexts such as operating systems, where such notions have typically been eschewed.

4. Conclusion

We feel that several basic conclusions can be drawn based on our involvement in the building of distributed languages and systems. The first is that research involving system or language design and implementation is hard. Our projects have consumed many resources, not only in the form of machine cycles, but also in the form of time spent performing the many extraneous tasks required by such work. However, the second conclusion is that despite these problems, experimental computer science is ultimately a rewarding activity. Our projects are, we believe, valuable contributions to research in the area of distributed languages and systems. In addition, our use of these projects as research catalysts has proved a valuable way to expand our investigations into other related areas. Finally, we point out

that many of the lessons outlined in Section 3 are general in that they apply to *any* system building effort and not just those based on a distributed model of computation. In other words, although there are certainly unique problems associated with distributed systems, it is clear that many of the issues in the construction of experimental systems are universal issues that must be addressed regardless of the architecture on which a system is based.

Acknowledgments

Many people have contributed to the design and implementation of projects surveyed in this paper, including A. Black, N. Buchholz, M. Coffin, I. Elshoff, R. Hayes, E. Jul, H. Levy, K. Nilsen, S. Manweiler, H. Pinnamaneni, T. Purdin, R. Raj, and G. Townsend. This work has been supported at the University of Arizona by the National Science Foundation under grants DCR-84-02090 and DCR-86-09396 and by the Air Force Office of Scientific Research under grant AFOSR-84-0072; Emerald was supported at the University of Washington by the National Science Foundation under grant DCR-84-20945. Equipment was provided at the University of Arizona by the DoD University Research Instrumentation Program (URIP) under grant AFOSR-85-0089 and the NSF Coordinated Experimental Research (CER) program under grant DCR-83-20138, and at the University of Washington by a Digital Equipment Corporation External Research Grant.

References

[Alme85] Almes, G.T., Black, A.P., Lazowska, E.D., and Noe, J.D. The Eden system: A technical review. *IEEE Trans. on Softw. Eng. SE-11*, 1 (Jan. 1985), 43-59.

[Andr85] Andrews, G.R. and Olsson, R.A. Report on the distributed programming language SR. TR 85-23, Dept. of Computer Science, The University of Arizona, Nov. 1985, revised Sept. 1987.

[Andr86] Andrews, G.R. and Olsson, R.A. The evolution of the SR language. *Distributed Computing*, vol. 1, no. 3 (July 1986), 133-149.

[Andr87a] Andrews, G.R., Schlichting, R.D., Haves, R., and Purdin, T.D.M. The design of the Saguaro distributed operating system. *IEEE Trans. on Softw. Eng. SE-13*, 1 (Jan. 1987), 104-118.

[Andr87b] Andrews, G.R., Olsson, R.A., *et al.* An overview of the SR language and implementation. *ACM Trans. on Prog. Lang. and Systems*, to appear.

[Atki87a] Atkins, M.S. Dealing with circularity in concurrent systems: upcalls vs. downcalls. Submitted for publication.

[Atki87b] Atkins, M.S. and Olsson, R.A. Performance of multi-tasking and synchronization mechanisms. Submitted for publication.

[Blac85] Black, A. Supporting distributed applications: Experience with Eden. *Proc. 10th Symp. on Op. Sys. Principles*, Orcas Island, WA (Dec. 1985), 181-193.

[Blac86] Black, A., Hutchinson, N., Jul, E., and Levy, H. Object structure in the Emerald System. *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR (Oct. 1986), 78-86.

[Blac87] Black, A., Hutchinson, N., Jul, E., Levy, H., and Carter, L. Distribution and abstract types in Emerald. *IEEE Trans. on Softw. Eng. SE-13*, 1 (Jan. 1987), 65-76.

[Cher85] Cheriton, D.R. and Zwaenepoel, A.W. Distributed process groups in the V Kernel. *ACM Trans. on Computer Systems 3*, 2 (May 1985), 77-107.

[Come86] Comer, D.E. and Peterson, L.L. Conversation-based mail. *ACM Trans. on Computing Systems 4*, 4 (Nov. 1986), 299-319.

[Gris83] Griswold, R. and Griswold, M. *The Icon Programming Language*. Prentice-Hall, Englewood Cliffs, N.J. 1983.

[Haye87a] Hayes, R. and Schlichting, R.D. Facilitating mixed-language programming in distributed systems. *IEEE Trans. on Softw. Engr.*, to appear.

[Haye87b] Hayes, R., Manweiler, S., and Schlichting, R.D. A simple system for constructing distributed, mixed-language programs. Submitted for publication.

[Haye88] Hayes, R. UTS: A type system for facilitating data communication. Ph.D. Dissertation, Dept. of Computer Science, The University of Arizona, in preparation.

[Hutc87] Hutchinson, N. Emerald: An object-based language for distributed programming. Ph.D. Dissertation, Dept. of Computer Science, The University of Washington, Jan. 1987.

[Jul87] Jul, E., Levy, H., Hutchinson, N., and Black, A. Fine-grained mobility in the Emerald System. *ACM Trans. on Computer Systems*, to appear.

[Jul88] Jul, E. Object mobility in Emerald. Ph.D. Dissertation, Dept. of Computer Science, The University of Washington, in preparation.

[Lamp78] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (July 1978), 558-565.

[Manw86a] Manweiler, S.W., Hayes, R. and Schlichting, R.D. The MLP system user's manual. TR 86-4, Dept. of Computer Science, The University of Arizona, February 1986.

[Manw86b] Manweiler, S.W., Hayes, R. and Schlichting, R.D. Adding new languages to the MLP system. TR 86-9, Dept. of Computer Science, The University of Arizona, June 1986.

[Olss86] Olsson, R.A. Issues in distributed programming: The evolution of SR. Ph.D. Dissertation, Dept. of Computer Science, The University of Arizona, Aug. 1986.

[Pete87] Peterson, L. Preserving context information in an IPC abstraction. *Proc. of the 6th Symp. on Reliability in Distributed Software and Database Systems*, Williamsburg, VA (March 1987), 22-31.

[Purd87a] Purdin, T. Enhancing file availability in distributed systems (the Saguaro file system). Ph.D. Dissertation, Dept. of Computer Science, The University of Arizona, Aug. 1987.

[Purd87b] Purdin, T., Schlichting, R.D, and Andrews, G.R. A file replication facility for Berkeley UNIX. *Software—Practice and Experience*, to appear.

[Schl86] Schlichting, R.D., Andrews, G.R, and Purdin, T. Mechanisms to enhance file availability in distributed systems. *Proc. 16th Int. Symp. on Fault-Tolerant Computing*, Vienna (July 1986), 44-49.

[Teit81] Teitelbaum, T., and Reps, T. The Cornell Program Synthesizer: A syntax-directed programming environment. *Commun. ACM 24*, 9 (Sep. 1981), 563-573.