Concert/C: Supporting Distributed Programming with Language Extensions and a Portable Multiprotocol Runtime

J. S. Auerbach A. S. Gopal M. T. Kennedy
J. R. Russell
IBM T. J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598
{jsa,ajei,mtk,jrussell}@watson.ibm.com

Abstract

We describe the design and implementation of the Concert/C compiler and runtime. We describe solutions to the problems of (1) how to extend a language without compromising the use of legacy source and object code and tools, (2) how to extract language-neutral interface information from native language type declarations, (3) how to make function pointers into first-class values transmissable over a network while retaining runtime compatibility with plain C, and (4) how to interoperate with multiple RPC and messaging protocols, selecting protocols at runtime, with little sacrifice of efficiency.

1 Introduction

The Concert/C language extends the ANSI C language [1] to enable the construction of *explicitly distributed* programs. The Concert/C language and its design rationale is explored in [6]. This paper deals with the implementation of the Concert/C compiler and runtime.

A program is "explicitly distributed" if it is aware of its use of distributed services. For example, an inter-enterprise billing system is explicitly distributed because its components represent autonomous enterprises. Network management is an explicitly distributed problem. Some "number crunching" programs are (explicitly) conscious of the number of computational nodes they have available. In contrast, *implicitly* distributed programs simply depend on a fixed number of remote "servers." These are easier to support.

In [6], we argued that supporting these kinds of programs means (1) ***hiding complexity***, (2) ***accommodating legacies***, both of program code and of programmer skills, and (3) ***accommodating heterogeneity***. We showed that the first two of these goals are in conflict: to achieve simplicity we need new programming languages, but accommodation of legacies requires that languages not be changed. The Concert/C compromise is to modify the programming language very cautiously, remaining a strict superset of ANSI C, and maintaining strict object code compatibility with existing compilers. Concert/C overloads such familiar concepts as function pointers so that they serve to express explicit distribution by "pointing" to functions in remote address spaces.

Having designed the Concert/C language, we were faced with a number of implementation challenges. It is one of the purposes of this paper to highlight these challenges and show how they were overcome. In addition, [6] did not show how Concert/C addresses the third goal of accommodating heterogeneity, since that is largely a matter of implementation. We will show in this paper how the Concert/C implementation uses novel techniques to accommodate heterogeneity in protocols and languages.

2 The Concert/C language

As described in [6], Concert/C provides for (1) transparent remote procedure call (RPC), (2) asynchronous messaging, (3) concurrency and synchronization (4) dynamic process creation, and (5) distributed linking. We will focus on the first two of these extensions, showing how each makes use of our generalized notion of pointers. We will touch on the other extensions as necessary to make examples complete.

Transparent RPC. Concert/C generalizes the concept of a C function pointer so that RPCs are made without special parameters for expressing remoteness. Function pointers become first-class values that can be transmitted to other processes in messages or RPCs, after which they "point" across process boundaries. This permits applications to control their logical connectivity explicitly. In addition, a process that creates another process automatically gets a pointer to a desig-

1063-6927/94 $03.00 © 1994 IEEE

nated **initial** function in its child. Finally, a program can obtain an initial set of pointers to functions in other programs through "distributed linking," a distributed reference-resolution process analogous to "link editing." The following example, consisting of three related Concert/C programs, illustrates these points.

```
/* Program A */
port int adder(int i, int j)
{
  return i + j;
}
[[ use shared_file {exports ADDBINDINGS};
  export adder; ]]
main()
{
  accept(adder);
}
```

Program A exports the function **adder** for distributed linking by other programs, and then accepts exactly one call to **adder**. The **port** keyword is what makes **adder** exportable, the distributed linking section delimited by [[ ]] provides export details, and the **accept** operator performs a rendezvous with whatever program calls **adder**.

```
/* Program B */
int (*adder)(int i, int j);
[[ use shared_file {imports ADDBINDINGS};
  import adder;
  program C "C-prog" newspace; ]]
main()
{
  void (*childs_init)(int (*)(int, int));
  create(C, &childs_init);
  childs_init(adder);
}
```

Program B creates an instance of program C, obtaining a pointer to program C's initial function. It then calls that initial function, passing the address of A's **adder** function (which it imported) as an argument.

```
/* Program C */
int (*adder)(int i, int j);
initial my_init(int (*afunc)(int, int))
{
  adder = afunc;
}
main()
{
  int answer;
  accept(my_init);
  answer = adder(2, 2);
}
```

In program C, the **initial** keyword designates **my_init** as C's initial function. The program starts by accepting a call to **my_init**, obtaining a pointer to A's **adder** which is stored in a global variable. Then, (in the **main** function) program C makes an RPC to A, passing it two numbers to add, and receiving the answer.

Messaging. In addition to RPC, Concert/C has support for simple message passing, which is sometimes a more natural style for expressing event-driven programs. Message passing and RPC may be freely mixed in a single program. To use message passing, the programmer declares a "receiveport" of any type. The program possessing the receiveport can **receive** from it. A program possessing a pointer to a receiveport can **send** to it. Receiveport pointers are first-class in the same sense as are function pointers. In the following example, program B sends the value 2 to program A, which receives it into **i**.

```
/* Program A */
receiveport {int} intq;
[[ /* exports intq */ ]]
main()
{
  int i;
  receive(intq, &i);
}

/* Program B */
receiveport {int} *intqp;
[[ /* imports &intq as intqp */ ]]
main()
{
  send(intqp, 2);
}
```

In addition to integers and function pointers, any ANSI C type can be used as an RPC function parameter or receiveport message type. Arithmetic types, enumerations, structures, and pointers to these types can be used directly. Unions, arrays, and aliasing pointers require the programmer to add *annotations* to the type declarations. Normally, a transmission involving a pointer "carries along" the thing pointed to (a "deep copy" is made) unless this behavior is overridden with an annotation.

We make available a complete specification of Concert/C [5] couched as a formal extension to [1], a tutorial [17], and a programmer's manual [4]. Also, a complete Concert/C system is available by anonymous ftp for experimental use.

3 The Concert/C compiler

One implementation problem we faced was how to be a superset of ANSI C without disrupting the use of ANSI C compilers, tools, debuggers, object code, etc. We chose to implement Concert/C as a "read mostly" preprocessor.

The compiler presents itself to the user as a driver module (**ccc**) that sequences the phases of compilation and linking. First, **ccc** runs each source file through a standard ANSI C pre-processor; this is possible because Concert/C uses ANSI lexical rules. The output of the ANSI preprocessor then passes through the Concert/C preprocessor **ccpp**, and **ccpp**'s output is processed by a standard ANSI C compiler. The standard system linker and standard ANSI C libraries are used to produce an executable.

Wherever possible, **ccpp** passes plain ANSI C code directly into its output stream, examining it without modifying it. User names are never changed, and user statements are not changed unless they contain Concert/C extensions. Concert/C type annotations are simply removed; declarations are otherwise unperturbed. Completely new types like **receiveport** are expanded into structures. Concert/C executable operations (like **create** and **accept**) are expanded into library function calls with some additional arguments. Line numbering and other information required for debugging is carefully preserved.

The main thing that **ccpp** does is to extract full type information for all types that may be transmitted remotely. The presence of a **port** function, a **receiveport** declaration, an invocation of **create** (which creates a remote pointer) or of a distributed linking directive triggers a "signature analysis" phase and the generation of stubs to perform marshalling and demarshalling. All stub output is captured in static tables that are placed in a "logical source file" different from the user's source file (the ANSI #line directive may be used to create such logical source files within a single physical source file); thus, the user sees only his program when using a symbolic debugger. However, these static tables are "in scope" and can be referenced in the macro expansion of Concert/C operators. We discuss the details of this signature analysis below.

4 Concert's approach to language independent types

Although only Concert/C presently exists, the Concert project [3, 34] contemplates a family of similarly extended languages (Concert/Fortran, Concert/Cobol, etc.) that are intended to be mutually interoperable. Such a multilanguage distributed computing tool must deal with the differences between the type systems of languages, and the Concert/C design takes this into account.

The usual solution to multilanguage interoperability is to provide a separate interface definition language (IDL) [26, 31, 18, 9]. Interfaces are specified in the IDL, and a specialized IDL (or "stub") compiler generates the code ("stubs") to marshall and unmarshall values for transmission. Conversion to and from a language-neutral format is embodied in those stubs.

Concert/C does ***not*** use a separate IDL, but relies as much as possible on the existing C type system, extending it with annotations where strictly necessary. As we show in [6], separating the IDL from the programming language weakens support for explicitly distributed applications. However, integrating stub compilation with the programming language requires that the problem of interlanguage interoperability be addressed in some other way, since each Concert language will have a different type system. There must be some way to determine unambiguously the type-compatibility of signatures expressed in different Concert languages. Also, an interface originally declared in one Concert language must be able to be translated to another, e.g., so that a project begun using only Concert/C can be extended using Concert/Fortran. The Concert signature representation (CSR) is our solution. The details of CSR are described in [7].

CSR is an intermediate language produced by each Concert compiler. It serves the function of an IDL in defining interoperability. It can mediate translation between interfaces expressed in different languages. It is designed for machine processing, not human comprehension. It is also a convenient notation from which to do stub compilation, permitting all concert languages to be supported by a single stub compilation back-end. In addition, because CSR is efficiently machine readable, it can be interpreted at runtime by a "universal stub" capable of marshalling any message. While such an "interpretive marshaller" is less efficient than compiled stubs, it provides a superior debugging environment, is easier to modify (and hence valuable when conducting experiments in protocol design), and can more readily support dynamically typed languages. We now have both a stub compiler (for efficiency) and an interpretive marshaller (for debugging and development). A simple switch passed to the compiler selects which to use.

Since there was no requirement that CSR be easy for programmers to write, we designed CSR with a different goal, that of providing maximum feasible interoperability between unlike programming languages. We do this by analyzing each function or receiveport signature into two parts. The *contract* is the part of the CSR that captures interoperability. The *endpoint modifier* is the "remainder" needed, along with the contract, to generate correct stubs for the present language. A single contract may have many different endpoint modifiers, and two processes need only agree on the contract to communicate successfully. By making the contract language as minimal as possible (in a sense that is explored in [7]), we give maximum flexibility for finding common ground between languages.

No matter what syntax the programmer uses to describe an interface, it has to do two things. It has to define what is needed for interoperability, and it has to guide stub generation for a particular language. The former goal requires that concrete details (such as the difference between **int**, **int *** and **int ****) be ignored, but the latter goal requires that such details be accurately recorded. Humans have trouble separating those two issues, and convenience dictates that information should only have to be specified once. So, no human-readable syntax can achieve good language interoperability without some postprocessing by the compiler. In Concert languages, we have the *compiler* do the work of separating the contract from the endpoint modifier, and only ask the human programmer to annotate his declarations to the extent necessary to enable the separation. For example, if a Concert/C programmer declares **int *[required] i**, he is stating that the pointer **i** cannot be NULL; hence, contractually, an integer is transmitted. This type is contractually compatible with a simple **int**. If he declares **int *[optional] i**, he is stating that the pointer **i** may be NULL; hence, contractually, a *choice* between an integer and an "empty" value is transimitted. This type is contractually compatible with **unions** in C, **variants** in some other languages, and, of course, pointers in those languages that have them. More details are presented in [7].

A CSR does not contain everything needed to do marshalling and unmarshalling: specifically, the communications protocol is not known until runtime. However, at compile time we know a *set* of possible protocols that we are prepared to support. Having first built a CSR, the compiler generates complete stubs for many different protocols. The results of these compilations are gathered in a structure called a "marshall plan," a collection of pointers to marshalling functions from which an appropriate selection is made at runtime based on the actual protocol (there is further discussion of the marshall plan in section 6). The CSR itself is saved in the marshall plan if interpretive marshalling is requested. Finally, a contract id (which is a hash value of the contract) is computed to use in runtime type checking.

5 Implementing "network" function pointers

Making function pointers able to "point to" functions in other machines and address spaces was critical to our complexity hiding strategy. Conceptually, a first-class "network" function pointer can be implemented by changing the usual C representation for function pointers so that it becomes a union of local and remote cases. The local case is still a machine address. In the remote case, the representation contains information on the identity of the remote address space, how to communicate with it, and how to tell it what function to call. We call this information *binding data.*

We did *not* change the actual representation of function pointers because that would limit the accommodation of legacy C code. By preserving the representation, we allow Concert/C function pointers to be passed to legacy code, enabling "implicit RPCs" by such code without recompilation. So, we employ short code fragments called *ministubs.* Local function pointers are unchanged: they point to the target function; remote function pointers point to ministubs in the local address space. Each ministub, when executed, retrieves its associated binding data and invokes a generic *RPC dispatcher*, passing it the the binding data plus the original call parameters. The RPC dispatcher is what converts the call into an RPC.

Since the number of RPC *types* in a program is statically known, marshall plans (see section 4) can be created statically for all of them. Since the number of function pointers to be imported by a process cannot be statically known, ministubs must be dynamically allocated. Such dynamic allocation requires an architecture-dependent solution for each target system. The Concert *ministub manager* has now been implemented for the RS/6000, the Sparc, 32 bit members of the Intel 80x86 family, and the IBM 390. There are, broadly speaking, two kinds of solutions.

For systems in which function pointers point to descriptors rather than directly at code, ministubs can be allocated without copying any code. A new descriptor is allocated pointing to a fixed ministub code body but also containing a pointer to the correct binding data. The linkage convention will load this pointer into a register as part of making a function call to the ministub; the ministub merely has to retrieve the information from the register. This approach was used on the RS/6000.

For systems in which function pointers point directly at code, ministubs are implemented by copying a code template. The binding data is placed at a fixed offset from the copied code, and the copied code captures the address of the data before branching to the RPC dispatcher. This approach was used for the Sparc, Intel, and 390. This can only work when the machine architecture permits code to placed in the data space.

Both approaches requires some machine-dependent code to be written, but they are efficient, reliable, and well worth the small porting effort. Systems that have direct function pointers and completely disjoint instruction and data spaces could force the use of less efficient techniques such as dynamic loading, or static preallocation of a pool of ministubs in conjunction with a hash table. We have not yet encountered a system that actually has this problem.

The binding data associated with each ministub contains two important fields: (1) the marshall plan, and (2) the *address list*. All binding data of the same signature can point to a single shared marshall plan, since the marshall plan is read-only. The address list contains a protocol-specific address for every protocol known to the runtime by which the remote process may be contacted. Each element in the list is a complete description of how to reach the target function. The ways in which the address list is used are taken up in the next section.

6 Protocol Heterogeneity

The Concert runtime provides an extensible framework for protocol heterogeneity. Concert processes communicate with each other via a private RPC protocol that can run over many different transports. This is not unusual in today's environment. However, Concert processes can also interoperate with non-Concert clients/servers via various "open" RPC protocols. It is straightforward to add new protocols to the Concert runtime. These may be either new transports for use by the Concert RPC protocol or new RPC protocols for interoperation with non-Concert clients and servers.

As discussed in the previous section, a function pointer that points outside its address space uses a ministub to recover hidden "binding data" at call time. A pointer to a receiveport in another address space points (directly) to the same kind of binding data. A key element in the binding data is an address list, which describes all possible way to reach the target function or receiveport. When either of these kinds of pointer is sent in a message or as an RPC parameter, it is the full address list that is sent. So, such a pointer "drags with it" complete information about all ways to reach the thing pointed to.

The runtime is able to interact uniformly with all RPC protocols (Concert and non-Concert) because it abstracts each one into a data structure called a *protocol handle*, which is a vector of functions. The functions perform the basic operations of sending, forwarding, replying and calling that are fundamental to the semantics of RPC and message passing. They understand the specific address format of whatever RPC protocol they implement, and can find the correct element in a Concert address list.

Similarly, the Concert RPC protocol interacts uniformly with various underlying transport protocols by using a *transport handle*, a function vector that performs the operations of connecting, sending, and receiving that are fundamental to the semantics of transport.

Introducing a new protocol is mostly a matter of wrapping some "veneer" code around an existing implementation of the protocol so that the result fits the protocol handle or transport handle model imposed by the Concert runtime. The present runtime has the Concert RPC protocol running over TCP (for network communication) and various intra-machine transports (for local efficiency). It uses the SUN ONC RPC protocol to interoperate with SUN ONC clients and servers. We are presently adding support for the OSF DCE RPC protocol (to interoperate with DCE clients and servers). The main reason why we have a private RPC protocol of our own is to support advanced Concert features such as first-class function pointers.

The choice of which protocol is to be used from the set present in an address list is made as late as possible, generally the first time communication from one address space to another is required. Once a connection exists, it will be reused. Not only is there connection caching at an address-space level (which requires a hash table), but a direct pointer is cached in each binding data structure, so that uses of a function or receiveport pointer after the first generally avoid even the hash table overhead. When a function or receiveport pointer is transmitted, the address list is transmitted but cached connection information, of course, is not; consequently, the receiving process is free to make an entirely different protocol choice. To illustrate, suppose process P invokes a function pointer `fn` that points to a function in process Q running on a different machine. The protocol chosen by P's runtime will be one that uses a network transport. Now suppose P **sends** `fn` to another process R that resides on the same machine as Q. When R invokes `fn`, its runtime will choose one of the intra-machine transports so that Q and R can interact efficiently. This is not the same protocol that P and Q used in their interaction.

Some address lists contain a mixture of Concert and non-Concert RPC protocols. These lists occur when the callee process is written in Concert/C. If the callee is a non-Concert program, however, the address list typically contains exactly one entry: that for the RPC protocol that the program understands. Between Concert programs, the Concert RPC protocol is always used and the transport is chosen based on proximity of the processes in order to maximize efficiency. When one of the partners is not a Concert program, the right RPC protocol is chosen because it is the only choice available.

It may not be obvious how a Concert program obtains a pointer to a function in a non-Concert program. Such pointers arise through distributed linking import declarations that specify use of an "open" directory service. For example, a Concert program can import a pointer from the SUN ONC portmapper. Such a pointer *might* have been placed in the portmapper by another Concert program or by an ordinary SUN ONC program ([6] shows an example of this). Because of its origin, its address list has only the SUN ONC address element.

The Concert RPC protocol uses a characteristic header called a *call package*. It contains the precise identity of the remote procedure to be invoked, a contract id (described in section 4) used for type checking, and an address list that enumerates all the protocols along which it can receive the results of the call (so that the call can be forwarded, as described below). After sending the call package header, the Concert RPC protocol marshalls the actual message or RPC parameters onto the wire, guided by the marshall plan taken from the binding data.

We identify four marshalling stages: marshalling the call by the caller, unmarshalling the call by the callee, marshalling the reply by the callee and unmarshalling the reply by the caller (asynchronous messaging has only two stages, marshalling by the sender and unmarshalling by the receiver). The marshall plan consists of a table. Each row of the table points to four stubs, one for each of the marshalling stages, and there is one row for each of the RPC protocols supported by the runtime. The protocol manager locates the row appropriate for the protocol, selects the column appropriate for the present marahalling stage, and and calls the stub whose address is found in that "cell" giving the actual data and some connection state as arguments.

When the callee's runtime receives the call package, it locates the marshall plan for the invoked function, and compares the contract id recorded there with that in the call package in order to guard against runtime type errors. It then makes a marshall plan row and column selection as described above and unmarshalls the message or call parameters. The call package itself also serves as an envelope for information to be placed on the target queue. The **receive** or **accept** operator will dequeue the call package, discard the call package itself, and do the right thing with the data.

Instead of handling the call, the callee may *forward* the call to a third process (the forwardee). In this case, the address list sent by the caller as part of the call package is also forwarded, thereby permitting the forwardee to reply directly to the caller, perhaps using a different protocol.

When the target of a call, send, or reply is a non-Concert program, the runtime's protocol handle for the RPC protocol does some of the same things, but does not generate a call package. Instead, it uses whatever protocol header is appropriate for the specific protocol is used. Conversely, when a non-Concert client communicates with a Concert process, the protocol handle code for that RPC protocol converts any incoming header information into a Concert call package so that it can be interpreted by the rest of the runtime. This constructed call package includes an address list containing a single entry: the return address according to the non-Concert protocol.

7 Related Work

Concert contains both a language and a system component, and as such can be compared with many other efforts. Unfortunately, space restrictions force us to limit such comparisons.

We choose the "multiple compatible language extensions" to best satisfy the conflicting goals of accommodating legacies and hiding complexity. An entirely new language designed to support distributed computing (*e.g.*, NIL [30], Argus [20], SR [2], Emerald [11], Hermes [29]; a survey may be found in [8]) can effectively and elegantly hide complexity for someone familiar with the language. However, such languages are, in general, poor at accommodating legacy code, and require new programmer skills.

At the opposite extreme, commercial packages (such as OSF/DCE [26], Apollo NCS [19], or SUN/RPC [31])) and software tools (such as Matchmaker [18], Courier [33], Horus [16], and HRPC [9]) do a good job of accommodating legacies, and sometimes heterogeneity. The stub compilers associated with these systems also hide *some* of complexity associated with marshalling data. However, as we discuss in [6], library-based tools do a poor overall job of hiding complexity.

Tools attempting to address the residual complexity of library-based packages [25, 24] typically generate a prologue to establish a program's initial connectivity in addition to the usual marshalling stubs. Such tools are best at making simple "clients" that use a relatively static set of "servers." They do not help when programs have an evolving relationship, nor do they simplify the writing of "servers."

Others (*e.g.*, Linda [15] and in some ways PCN [13]) have also chosen multiple language extensions as their basic approach. The Mercury project [21] sought to achieve language interoperability by grafting a new model of communication onto existing languages and mapping their data types to a common model of transmissible data. All these efforts share with Concert the tradeoff between simplicity and compatibility, and also share a similar starting point for accommodating language heterogeneity. Concert differs from these other efforts in the accommodation of *protocol* heterogeneity. As we described, Concert programs interoperate with programs not written in Concert/C. In contrast, the "glue" provided by Linda, PCN, Mercury, *etc.* only binds together programs which knowingly use the specific extensions provided by those tools.

Most RPC products today are "multiprotocol" at the transport level, as is Concert, but few are "multiprotocol" at the RPC level. This attribute Concert shares only with HRPC [9]. Several things differentiate Concert/C from HRPC. Because it gets less help at translation time, HRPC uses a largely interpretive approach at runtime, in which explicit calls must be made through three layers of protocol support (control, data representation, and transport) in order to resolve the protocol. Concert/C pre-compiles for a set of target protocols at compile time and makes a single selection at runtime, after which all marshalling and demarshalling may be done by compiled code. In HRPC, once a protocol is selected for a particular RPC handle, that protocol is fixed. In Concert/C, as a first-class pointer to a function or queue is passed around the network, it retains knowledge of *all* protocols capable of reaching the process which defines the function or queue. Different protocol selections may be made by different senders at different times in the lifetime of the binding.

8 Status, Usage, and Future Work

Currently, Concert/C runs on AIX 3.2 on Risc System/6000s, Sun OS 4.1 and Solaris on Sparc stations, and OS/2 2.1 on IBM PCs, with communication over TCP and OSF/DCE and interoperability with SUN RPC.

Some serious applications have been built with Concert/C. One application searches a database of genetic information on 64 Risc System/6000s in parallel [12]. Another uses similar search techniques to perform complex visual pattern recognition [28]. Concert/C has been used to connect a natural language textual query program to a collection of large, geographically dispersed text repositories [23]. Another project uses Concert/C to enable the graphical interconnection of running applications, thus enabling cooperative computing over a network [22].

The performance of the current prototype is comparable to that of Sun RPC. Parsons [27] has compared the usability and performance of Concert/C with other tools for distributed programming including Isis [10] and PVM [32].

We are adding full interoperability with OSF/DCE, are currently designing Concert/C++, and are investigating preliminary designs for Concert/Fortran. We are interested in developing tools for process management and control, such as a distributed visualizer/debugger, and utilities which aid in the design of interoperable interfaces, using our CSR as an intermediate form to mediate the search for equivalent representations in different languages. We are also investigating extensions to the language to increase its effectiveness in areas such as group communication.

References

[1] American National Standards Institute. *Programming Language – C*, ANSI/X3.159-1989 edition, 1989.

[2] Gregory R. Andrews. Synchronizing Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, October 1981.

[3] J. S. Auerbach, D. F. Bacon, A. P. Goldberg, G. S. Goldszmidt, A. S. Gopal, M. T. Kennedy, A. R. Lowry, J. R. Russell, W. Silverman, R. E. Strom, D. M. Yellin, and S. A. Yemini. High-level language support for programming distributed systems. In *1992 International Conference on Computer Languages*, pages 320–330. IEEE Computer Society, April 1992.

[4] Joshua Auerbach, Arthur P. Goldberg, German Goldszmidt, Ajei Gopal, Mark T. Kennedy, Josyula R. Rao, and James R. Russell. Concert/C manual: A programmer's guide to a language for distributed C programming. Technical Report RC19332, IBM T. J. Watson Research Center, 1993.

[5] Joshua Auerbach, Arthur P. Goldberg, German Goldszmidt, Ajei Gopal, Mark T. Kennedy, James R. Russell, and Shaula Yemini. Concert/C specification: Definition of a language for distributed C programming. Technical Report RC18994, IBM T. J. Watson Research Center, 1993.

[6] Joshua S. Auerbach, Arthur P. Goldberg, German S. Goldszmidt, Ajei S. Gopal, Mark T. Kennedy, Josyula R. Rao, and James R. Russell. Concert/C: A language for distributed programming. In *Winter 1994 USENIX Conference*, 1994.

[7] Joshua S. Auerbach and James R. Russell. The Concert Signature Representation: IDL as intermediate language. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Interface Definition Languages*, 1994.

[8] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming languages for distributed computing systems. *ACM Computing Surveys*, 21(3), September 1991.

[9] B. N. Bershad, D. T. Ching, E. D. Lazowska, J. Sanislo, and M. Schwartz. Remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering*, 13(8):880–894, August 1987.

[10] Kenneth P. Birman, Robert Cooper, et al. The ISIS system manual, version 2.0. Technical report, CS Department, Cornell, March 1990.

[11] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[12] A. Califano and I. Rigoutsos. FLASH: A Fast Look-Up Algorithm for String Homology. In *Proceedings First International Conference on Intelligent Systems for Molecular Biology*, Washington, DC, July 1993.

[13] K. M. Chandy and S. Taylor. The composition of concurrent programs. In *Proceedings Supercomputing '89*. ACM, November 1989.

[14] D. Gelernter and N. Carriero. Applications experience with LINDA. *SIGPLAN Notices*, 23(9):173–187, September 1988.

[15] Phillip B. Gibbons. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering*, SE-13(1):77–87, January 1987.

[16] Arthur P. Goldberg. Concert/C tutorial: An introduction to a language for distributed C programming. Technical Report RA218, IBM T. J. Watson Research Center, 1993.

[17] Michael B. Jones and Richard F. Rashid. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. Technical Report CMU-CS-87-150, CS Department, CMU, September 1986.

[18] Mike Kong, Terence H. Dineen, Paul J. Leach, Elizabeth A. Martin, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. *Network Computing System Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[19] B. Liskov. Distributed programming in Argus. *Comm. ACM*, 31(3), March 1988.

[20] B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury system. *Proceedings of the 21st annual Hawaii International Conference on System Sciences*, pages 178–187, January 1988.

[21] Andy Lowry, Rob Strom, and Danny Yellin. The Global Desktop, IBM T. J. Watson Research Center. To be published, available from the authors.

[22] Y.S. Maarek, D.M. Berry, and G.E. Kaiser. An information retrieval approach for automatically constructing software libraries. *Transactions on Software Engineering*, 17(8), August 1991.

[23] Netwise. *C Language RPC TOOL*, 1989. Boulder, Colorado.

[24] NobleNet. *EZ-RPC Manual*, 1992. Natick, Ma.

[25] Open Software Foundation, Cambridge, Mass. *OSF DCE Release 1.0 Developer's Kit Documentation Set*, February 1991.

[26] Ian Parsons. Evaluation of distributed communication systems. U. Alberta. Available from author.

[27] I. Rigoutsos and R. Hummel. Distributed Bayesian Object Recognition. In *Proceedings IEEE Conference on Computer Vision and Pattern Recognition*, New York City, NY, June 1991.

[28] Robert E. Strom, David F. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Alexander Yemini. *Hermes: A Language for Distributed Computing*. Prentice Hall, January 1991.

[29] Robert E. Strom and Shaula Alexander Yemini. NIL: An integrated language and system for distributed programming. In *SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, June 1983.

[30] Sun Microsystems. *SUN Network Programming*, 1988.

[31] V. S. Sunderam. Pvm: A framework for parallel distributed computing. *Concurrency: Practice & Experience*, 2(4):315–339, December 1990.

[32] The Xerox Corporation. *Courier: The Remote Procedure Call Protocol*, December 1981. Technical Report XSIS 038112.

[33] S. A. Yemini, G. Goldszmidt, A. Stoyenko, Y. Wei, and L. Beeck. Concert: A high-level-language approach to heterogeneous distributed systems. In *The Ninth International Conference on Distributed Computing Systems*, pages 162–171. IEEE Computer Society, June 1989.