Concert/C: A Language for Distributed Programming

Joshua S. Auerbach *Arthur P. Goldberg* *Germán S. Goldszmidt*
Ajei S. Gopal *Mark T. Kennedy* *Josyula R. Rao*
James R. Russell

IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598

Abstract

Concert/C is a new language for distributed C programming that extends ANSI C to support distribution and process dynamics. Concert/C provides the ability to create and terminate processes, connect them together, and communicate among them. It supports transparent remote function calls (RPC) and asynchronous messages. Interprocess communications interfaces are typed in Concert/C, and type correctness is checked at compile time wherever possible, otherwise at runtime. All C data types, including complex data structures containing pointers and aliases, can be transmitted in RPCs.

Concert/C programs run on a heterogeneous set of machine architectures and operating systems and communicate over multiple RPC and messaging protocols. The current Concert/C implementation runs on AIX 3.2[1], SunOS 4.1, Solaris 2.2 and OS/2 2.1, and communicates over Sun RPC, OSF/DCE and UDP multicast. Several groups inside and outside IBM are actively using Concert/C, and it is available via anonymous ftp from `software.watson.ibm.com:/pub/concert`.

1 Introduction

This paper describes Concert/C, a new language for distributed programming. We describe the basic Concert/C features needed to write significant Concert/C programs. We also show example Concert/C programs, and present comparisons with traditional RPC technology to motivate our design choices.

The objective of the Concert project is to develop tools to improve the productivity of people writing distributed programs for use in today's rich networking environments. Since most workstation and PC programmers write in C, we chose to extend C. We added a minimal set of new concepts, types, and operators, and reused existing C features whenever possible, so that a C programmer could learn Concert/C easily. In addition, we made Concert/C programs link-compatible with existing C object code to avoid sweeping recompilation of legacy code.

In big organizations, most LANs connect a heterogeneous set of machines, including personal computers running DOS, Windows, OS/2 and NetWare, Macintoshs, and workstations running different flavors of Unix. A variety of protocols run over these nets, including SNA, TCP/IP, NetBios and SPX/IPX. We designed Concert/C to exist in this environment. Our compiler is portable to any system with an ANSI C compiler, and our "multi-protocol" run-time can communicate over a heterogeneous set of protocols.

We chose to implement Concert/C as a language, rather than a set of library functions like PVM [33] or a set of library functions plus a stub compiler like SUN ONC assisted by tools such as `rpcgen` [32]. We made this decision because language-integrated support for distributed programming can offer a higher level of functionality than other approaches, as we demonstrate in section 3.

[1] All trademarks appearing in this paper are recognized registered trademarks of their respective companies.

2 The Concert/C Language

Concert/C [3, 4] is a superset of ANSI C. Like an ANSI C program, a Concert/C program is built from separately compiled source files combined in a link step. Constructing it differs from constructing an ANSI C program only in the use of the Concert/C compiler (ccc) instead of an ANSI C compiler (e.g. cc) for those source files which contain Concert extensions.

Concert/C is fully link-compatible with ANSI C, so only a small fraction of the application will typically need to be compiled with ccc. Ccc is actually implemented as a preprocessor using ANSI C as its back end, which enhances portability, helps ensure object code compatibility, and allows ANSI C debuggers to be used. A Concert/C program is a normal executable, and may be placed into execution by all the usual means.

The design of Concert/C's extensions exploits the fact that ccc reads application source files containing both interface declarations *and* program logic. Although ccc does not transform the program logic (for example, Concert/C has no control-flow primitives not taken directly from ANSI C), it can find many errors and automate certain functions which it could not if it saw only interface declarations. We call this a *single-translator* approach because, ccc can translate anything that cc can (in practice, it does not always need to).

Our single-translator approach contrasts with that of "add-on" RPC packages (such as SUN ONC, Apollo NCS, or OSF DCE), which may be characterized as a *two-translators* approach. In that approach the C compiler reads all of the program's original executable logic while an IDL translator or stub compiler (for example, `rpcgen`), reads interface definitions. Neither translator can read the other's source files, so the original executable logic is joined with the stub compiler's "canned" logic for marshalling and demarshalling of messages only at link time. This limits the extent to which the tool can help the programmer.

We describe the features and language extensions of Concert/C below. These fall into six general categories. While add-on packages could theoretically provide some help in each of the areas listed below, most provide only RPC, a single method for exchanging bindings, and (in the more recent ones) a strategy for demultiplexing requests based on locally concurrent threads. Some provide messaging, but none that we know of provides any help with process management. A complete facility for distributed computing must cover all these areas.

Despite the fact that Concert/C uses a single-translator approach, not every feature is provided as a language extension; language extensions are introduced only where there is significant "mileage" to be gained. We publish explicit runtime library calls, or use preprocessor macros, for other cases where a language extension is not required. By convention, Concert/C language keywords have simple mnemonic names while library functions and macros have names beginning with `cn_`. The examples in section 3 will illustrate these in more detail.

1. *Transparent Remote Function Calls*: Concert/C transparently "overloads" the C function call construct to embrace remote procedure call (RPC). If there are several remote instances of the same interface, Concert/C uses different function pointer *values* as a way of determining to which instance a call is to be directed, rather than using a "handle" parameter for this purpose. In general, a pointer to a remote function serves as a *binding*, conveying the capability to invoke that function.

   Placing a function in the **port** or **initial port** storage class causes a message queue to be associated with the function, enabling it to be called remotely.

2. *Asynchronous Messaging*: Concert/C adds one-way, asynchronous, queued messaging to the language. By analogy to function pointers, a pointer to a remote queue serves as a binding, conveying the capability to enqueue a message.

   The **receiveport** { *message-type* } type constructor defines a queue of messages of type *message-type* (which may be any Concert/C type). The operation **send**(*binding, message*) sends *message* to the queue to which *binding* refers, and the blocking operation **receive**(*port, message-ref*) moves a message from a queue into the variable pointed to by *message-ref*.

3. *Process Management*: Concert/C adds process management to the language. The new types **process** and **program** describe Concert/C processes and programs. A Concert/C program can start another Concert/C program executing on the same or a different machine using the **create**(*program, binding-ref*) operation. The *binding-ref* argument is a pointer to a binding variable in the parent, which is initialized by **create** to provide a logical connection to the child (specifically, a pointer the the **initial port** in the child) which is immediately usable for either RPC or messaging.

   Create also returns a value of type **process**, which can later be used by the **terminate** operation to kill the created process.

4. *Distributed Linking*: Concert/C provides an extensible set of "External Binding Facilities" (EBFs) which can store and retrieve Concert/C remote pointers to and from external media of various kinds. Many popular methods of finding components in a distributed system are supported in this way, including "well-known" ports (like the ones listed in /etc/services), the SUN ONC portmapper, shared files, strings, and the OSF DCE cell-directory service. This becomes a vehicle both for evolving the connectivity of a Concert/C application, and also for interoperating with popular RPC-based components not written in Concert/C. When used in the latter way, EBFs transparently map between Concert/C's use of "remote pointers" to express logical connectivity and whatever method is used by the other component. EBFs are provided both in executable macro form and in a convenient declarative form which is executed automatically at program start time. In some cases, use of declarative EBFs removes the need to write a `main` function.

5. *Demultiplexing*: Concert/C provides the necessary demultiplexing primitives to permit a server to make progress in the face of arbitrary request arrival orders. Included in this category is the ability either to deal with an RPC in a "rendezvous" style as a procedure call or decompose it into "request" and "response". The blocking operation **accept**(*function-port-list*) automatically handles a call to one of the function ports in the list when it arrives. The blocking operation **select**(*queue-list*) and the non-blocking operation **poll**(*queue-list*) return the index of a queue in *queue-list* that has a message (**poll** returns 0 if no queue does). These can be used to schedule the message **receive**. The new type **cmsg**{*function-type*} holds the "request" message associated with a remote call. The operation **reply** issues the "response" to such a message, thus completing a decomposed call.

6. *Communication of Complex Data Types*: Concert/C permits arbitrary C data types, including scalars, arrays, unions, and data structures built with pointers and aliases, to be transmitted in interprocess communications. If the data being transmitted (as an argument to an RPC, or as a message) contains pointers, then marshalling will make a *deep copy* – following the pointers[2], and the pointers in the data they point to, and so on, copying the entire data structure for transmission.

   Marshalling and demarshalling some data types requires more information about the data than standard C type declarations provide. The Concert/C programmer can provide additional information by *annotating* the IPC data. Annotations can be placed on declarations of data types communicated between processes, and declarations and definitions of ports and bindings. The annotations are also called *attributes*.

   The use of attributes is unnecessary for "simple enough" functions. A function is "simple enough" if it uses no unions, arrays are limited to character strings, pointers don't introduce any actual aliases, and parameters pass information in the "obvious" directions ("in only" for non-pointers, both directions for pointer-indirected arguments).

   Attributes add several kinds of information to communications.

   - *Data description* attributes expand on C's description of data types. For example, they indicate the length of arrays, the presence of null-terminated character strings, and the case of unions.
   - *Communication direction* attributes indicate the transmission direction of remote function call parameters. An **in** argument is transmitted from the caller to the callee, an **out** argument from the callee to the caller, and an **in,out** argument is transmitted both ways.

[2] However, the annotation **opaque** can instruct the marshaller to not follow a pointer.

Concert/C Attributes

Attribute	*Meaning*
How much memory is allocated to hold an array?	
max_length(i)	memory for an array of i elements is allocated.
Which direction is an argument transmitted?	
in	argument is transmitted from caller to callee.
out	argument is transmitted from callee to caller.
in,out	argument is transmitted from caller to callee and from callee to caller.
What case is a union?	
switch_is(i)	identifier i indicates the case of the union.
switch_type(type_specifier)	type_specifier indicates the type of the union discriminant.
case(case_list)	case_list identifies the case the union is in when the discriminant has a value appearing in case_list.
Can a pointer reference an object that other pointers reference?	
aliased	yes. marshaller detects aliases. aliased data at sending end becomes aliased data at the receiving end.
unique	no. marshaller does not detect aliases.
How big an array should be transmitted?	
length(i)	an array has i elements.
string	following the C character string convention, the array is terminated by a null character.
Which allocated memory is freed by the marshaller?	
keep(caller)	memory in caller or sender is not freed.
keep(callee)	memory allocated in callee is not freed.
discard(caller)	memory in caller or sender is freed.
discard(callee)	memory allocated in callee is freed following return.
Might a pointer be NULL?	
optional	yes. check for NULL pointer when marshalling.
required	no. marshaller assumes pointer references data to transmit.

Figure 1: Concert/C attributes.

- *Storage retention* attributes help control the marshaller's use of heap storage. Demarshalling allocates heap memory to store some IPC data. Storage retention attributes control whether the marshaller de-allocates these data.

Attributes appear in the syntax as a type-qualifier, like **const** and **volatile**. They are given in a comma separated list contained in square brackets: [**attribute1, attribute2,...**]

The Concert/C attributes are summarized in figure 1.

3 Examples and Comparisons

In this section, we present some detailed examples of the Concert/C language. We will show how each extension is accomplished, and contrast it with what would be the case if we were limited to the two-translators approach followed by add-on RPC packages. To make the discussion concrete, we will use a single programming example (which we will elaborate as we go along), showing how Concert/C does things, and also how the same thing would need to be done using public domain SUN ONC with `rpcgen`.

We chose this technology for the comparison because it is familiar and widely available, so most readers will be able to try similar examples for themselves. In some areas, other packages (such as NCS or DCE) may be more functional than public domain SUN ONC; we will try to point these out. However, the limitations on which we will focus most heavily are intrinsic to the two-translators approach, and are true of *all* add-on RPC packages.

3.1 Transparent RPC, Single Server, Fixed Relationship

Our simplest example starts out with one client program and one server program. In the usual RPC style, we will express the server first as a subroutine to be called (perhaps) remotely. Its purpose is to provide either an error code (if it fails) or (if it succeeds) a linked list of structures which constitutes a report on machine resource utilization by user.

We started with this example because the two-translators approach does reasonably well with it. In fact, the two-translators approach tends to do well as long as a client must interact with only one server of a given type. Even here, however, Concert/C has some significant advantages, stemming from the programmer's ability to reason about types using one language rather than two, and from the simplified mechanics of program construction.

Conceiving the example first as a nondistributed, ANSI C program, its interface definition, contained in a header file (`example1.h`), might look something like the following.

```
enum  status { success, failure };
enum  reason { not_authorized, system_error };
struct user { /* list of users and their utilization, expressed as a list */
   struct user * next;
   char * name;
   unsigned long cpu;
   unsigned long memory;
   unsigned long disk;
};
union result {
   struct user * userlist;
   enum reason why_failure;
};
enum status get_utilization(union result *answer);
```

A client should look something like this.

```
#include "example1.h"
main()
{
  union result a;
  switch(get_utilization(&a))
    {
  case success:
     /* display result list anchored at a.user_list */
     break;
  case failure:
     /* report failure reason from a.why_failure */
    }
}
```

The server subroutine would look something like the following (omitting all the purely local logic).

```
#include "example1.h"
```

```
enum status get_utilization(union result * answer)
{
  /* try to gather the statistics (never mind how) */
  if (it_worked) {
    answer->userlist = what_we_gathered;
    return success;
  }
  else {
    answer->why_failure = why_it_failed;
    return failure;
  }
}
```

In Concert/C, to make this a distributed client-server application, we just take the header file, and annotate the definition of `union result` and the type of function `get_utilization`.

```
[switch_type(enum status)]
union result {
   [case(success)]
   struct user * userlist;
   [case(failure)]
   enum reason why_failure;
};
typedef enum status get_utilization_t(
   [out, switch_is(return)] union result *answer
                           );
```

We also changed the function definition to a type definition; this was not strictly necessary but is a convenience for later.

It is not necessary to recompile either the original client or the original server subroutine with `ccc`. Instead, a server main procedure is obtained by using `ccc` to compile the following separate Concert/C source file.

```
#include "example1.h"
extern port get_utilization_t get_utilization;
[[
  use shared_file;
  export get_utilization {to "somefilename"};
  automain;
]]
```

The first line is simply a declaration of the function `get_utilization` in terms of its type. The `port` keyword is a Concert/C extension which informs the compiler that the function in question might be called remotely. The double brackets set off a section containing the declarative form of EBF syntax. The `use` statement indicates which EBF is to be employed (the "shared file" EBF in this case), `export` invokes some EBF-specific logic to export a function or message queue pointer to an external medium, and `automain` requests an automatic main function which loops forever listening for requests to all exported functions.

In the client, the simplest way to bind client and server together is to provide the matching declaration and EBF syntax, again in a separate `ccc` source file.

```
#include "example1.h"
get_utilization_t get_utilization;
[[
```

```
  use shared_file;
  import get_utilization {from "somefilename"};
]]
```

Note that we didn't use the port keyword here, because this module is not going to have it's get_utilization called remotely from any other module; rather, get_utilization is to be resolved by importing from a shared file. Putting the Concert/C client extensions in a separate source file to avoid recompilation was only a suggestion. The Concert/C client and server extensions could just as easily have been placed in the same source files with the original client and server source.

In comparison, to provide the same functionality using the two-translators approach, one needs to supply a separate IDL file. The following example shows one for rpcgen. While it resembles the original header file, note that it differs from ANSI C syntax, and thus the programmer must construct it separately.

```
enum  status { success, failure };
enum  reason { not_authorized, system_error };
struct user {
   user * next;
   string name<>;
   unsigned long cpu;
   unsigned long memory;
   unsigned long disk;
};
union result switch (status discrim) {
   case success:
      user * userlist;
   case failure:
      reason why_failure;
};

program GET_U_PROG {
    version GET_U_VERS {
        result get_utilization (void) = 1;
    } = 1;
} = 0x31234567;
```

In contrast to Concert/C, the correspondence of IDL types in the IDL file to C types in the C source is no longer exact. The enums retained their ANSI C syntax, but we had to deviate from ANSI C when we got to the struct and the union. We needed the special type string. We also had to assign a *name* (discrim) as well as a type (status) to the union discriminant, because rpcgen will actually generate a structure with a union and the union's discriminant encapsulated inside it. The type name result will actually refer to that structure, not the union. In order to use what rpcgen generates, we have to understand that it has done this transformation (in general, the generated interface header file has to be read). We were forced to relinquish the use of the identifiers specified for the enum and struct tags since rpcgen has automatically declared them as typedef names. Finally, we have had to accept a compromise both on the name (it becomes get_utilization_1, incorporating the version number) and on the exact type signature of the remote function (it needs to return a single output and take into account the "encapsulated" form of union which rpcgen generates). As rewritten, it looks something like the following.

```
#include "example1.h" /* generated by rpcgen */
result * get_utilization_1(void)
{
  static result answer;
```

```
  /* try to gather the statistics (never mind how) */
  if (it_worked) {
    answer.result_u.userlist = what_we_gathered;
    answer.discrim = success;
  }
  else {
    answer.result_u.why_failure = why_it_failed;
    answer.discrim = failure;
  }
  return & answer;
}
```

The change was a modest one, but that is not our point. The programmer had to understand a number of things about the correspondence between the language of the stub compiler and the programming language in order to get the program into the correct form. This need for mental translation is avoided when the interface declaration language and programming language are more tightly integrated to begin with, as they are with a single-translator approach.

These difficulties are inherent in the two-translator approach. Other stub compilers may do better than rpcgen in making some of the steps convenient, or in providing more overall expressiveness in the IDL (for example, OSF DCE's IDL is sufficiently expressive to cover most things that C programmers want to do), but *all* add-on RPC packages require programmers to do some mapping between two *different* type systems. Also, all such packages require treating the IDL file as something requiring a special compilation step; it is not directly usable as include file. Mechanically, the stub compiler produces many outputs (a header file, and several source files, some of which need to be incorporated into client and some into server). The Concert/C case had none of those complexities, although it did require some declaration to be done (of course).

It happens that SUN ONC with rpcgen also requires an explicit "binding" call (and subsequent error handling) in the client program in order to produce a "client RPC handle." It also requires that that handle be passed to the client stub procedure, making the RPC considerably less transparent. We won't show that here, since other add-on RPC packages, such as OSF DCE, succeed in hiding this particular source of complexity (about as well as Concert/C does) for this simple one-server case. However, in the next section we will see that a small change to the problem will force explicit binding and some form of "RPC handle" to be used in all known RPC tools that use the two-translators approach.

A server main procedure is generated automatically by rpcgen just as it is for simple-enough cases with Concert/C.

Before leaving this simple example, we consider a variation: suppose that the server had already been written using SUN ONC and rpcgen, and we wish to call it from a Concert/C client. Concert/C does not require that the server be rewritten in Concert/C to accomplish this. To write the client, we recognize that the server was written using a technology that requires certain restricted forms for function signatures, and simply change our view to conform to its, by slightly modifying the Concert/C declaration.

```
struct result {
   enum status discrim;
 [switch_is(discrim)]
   union result {
 [case(success)]
   struct user * userlist;
 [case(failure)]
   enum reason why_failure;
     } u;
};
struct result * get_utilization(void);
```

Then, in the Concert/C client program we have the following EBF logic.[3]

```
cn_sunrpc_import(HOST, GET_U_PROG, GET_U_VERS,
   GET_U_PROC, 0, "udp", get_utilization);
```

HOST needs to be given some value (the host on which the appropriate SUN ONC server resides). GET_U_PROG, GET_U_VERS, and GET_U_PROC are constants for the program number, version number, and procedure number assigned by rpcgen. The 0 indicates that the portmapper is to be used (a nonzero here would be taken as a well-known port) and the next argument indicates which of SUN ONC's two primary protocols are to be used. The last argument is, of course, a reference to the function which is being imported. After executing this EBF macro, we can call the imported function get_utilization just as before. A similar approach could be used for the dual of this variation: where the client is written using rpcgen and the server using Concert/C.

An EBF macro such as is illustrated here, although provided as a preprocessor macro backed by a library function, could actually not be supported in such a convenient form without our single-translator approach; that is, it requires language extensions underneath. In the expansion of this macro, the primitive language extensions crt_marshall_plan(get_utilization) and crt_microstub(get_utilization) are employed. The Concert/C compiler, which knows the type of function get_utilization, responds to the first of these directives by generating a table of anonymous stubs (called a "marshall plan"). This avoids requiring any static name to be assigned to such stubs in the event that the type of the last argument were to be a function pointer. The value of this will be seen in the next section.

Similarly, because the compiler "knows" that that get_utilization is a function and not a function pointer, it responds to the second directive by providing a function body (known as a "microstub") which adjusts for the difference in calling mechanics between the two. This permits the underlying library function to always produce a function pointer result, making it more general.

Two key points are illustrated by this variation. First, it is possible to have the advantages of Concert/C while still interoperating with the non-Concert world; we use EBFs to express this. Indeed, one can use the sunrpc and osf_dce[4] EBFs in the same program, and interoperate with both SUN ONC and OSF DCE. Of course, to actually deliver interoperability with SUN ONC and OSF DCE components is also a challenge[5] for protocol management in the runtime and in stub compilation; we deal with this aspect of Concert/C in [5]. Second, supporting dynamic EBFs is greatly aided by having a single-translator approach. The compiler was able to discover the need to do stub compilation, and do it without assigning static fixed names to stubs; no separate stub compilation was needed.

3.2 Transparent RPC, Multiple Servers, Dynamic Relationship

Now let's change the problem slightly, in order to see where the power of Concert/C really pays off. Suppose we want to write a client which will invoke multiple instances of the server subroutine described in the previous section, each of these to execute on a different machine. Suppose, further, that the list of machines is given on the command line, and so not known statically. And suppose, finally, that we have not pre-planned which machines will need to run these servers, and so we have not arranged for them to be started automatically by inetd, would not register them with the portmapper, etc. In short, we want to start them as needed ("on the fly") and terminate them when we are finished.

To do this in Concert/C, we would not have to change the header file at all (beyond what we did already in the previous section), and we would not need to change the server subroutine either. We would discard the server ccc source containing automain, and use, instead, the following variation.

```
#include "example1.h"
```

[3] In the present Concert/C implementation, we have only an executable macro form for the "sunrpc" EBF, which is why we don't use the declarative syntax; such a syntax may be added soon.

[4] Under development.

[5] There is no uniform mechanism for handling external events within a Unix process; nor is there an API for routing event streams to the various API's which wish to consume them like the X Windows library, or the Sun RPC or OSF DCE runtimes.

```
initial port get_utilization_t get_utilization;
main()
{
  accept(get_utilization);
}
```

The `accept` operator provides the "rendezvous" facility for Concert/C. The main program illustrated above accepts exactly one call to `get_utilization` and then exits. By making `get_utilization` the `initial` port, we tell the compiler that not only will it be called remotely, but the Concert/C process which creates this process (the "parent") will automatically get a pointer to it. We lost the automatic main, but the main which we had to write was trivial. With any of the various two-translators approaches, we would also have to write a main procedure for this case, and it would be far from trivial.

The new client, now a full-fledged Concert/C program, looks something like this.

```
/* somewhere: define the server program name as SERVER_NAME */
#include "example1.h"
#include "concert.h" /* get the cn_ library functions and macros */
main(int argc, char *argv[])
{
  char * host;
  get_utilization_t * get_utilization_p;
  union result a;
  program pgm = cn_prog_init();

  cn_prog_set_name(pgm, SERVER_NAME);
  while (host = *(++argv))
    {
     cn_prog_set_host(pgm, host);
     if (create(pgm, &get_utilization_p) == cn_noprocess)
       { /* deal with create error */ }
     switch(get_utilization_p(&a))
       { /* like original client */
     case success:
        /* display result list anchored at a.user_list */
        break;
     case failure:
        /* report failure reason from a.why_failure */
       }
}
```

Recall that there are Concert/C extensions in the header file; these are still needed, of course. We have added two new ones. The `create` keyword provides a new primitive operator (`create` is not a library function, though its syntax mimics one). The `program` keyword provides a new data type, generally called a "program description." A program description is a complex reference to a program, designating the program's file name, the machine it is to run on, and various options for instantiating it. The various `cn_prog_` library functions are used to initialize program descriptions.

The `create` primitive puts a program into execution, yielding a value of type `process` (called a "process handle") as its expression value, and possibly mutating its second argument, which will receive a pointer to the `initial` function or queue in the created process. This pointer can be used to communicate with the "child." If we had retained the process handle in this example, we could have later used it used with the `terminate` operator to cancel the child's execution (by the "severest" available means). This was not necessary since the child program was written to terminate voluntarily after one call. The only use we made of the process handle in the example was to compare it to special process handle value `cn_noprocess` which would mean that no child was created (a richer exception handling facility exists in

Concert/C but is not discussed in this paper). We used the pointer to the child's initial function in order to make an RPC to the child.

What is illustrated here is the joint power of providing process dynamics along with communication, and also using function pointers (which are anonymous and re-assignable) to express remoteness. By using this approach, we were able to write a highly dynamic client which was not much more complicated than the original static client. Although it may not be obvious at first, a single-translator solution is needed to make a pointer-based approach work effectively. Because `create` is an operator and not a function, `ccc` is involved in analyzing the type of its second argument, and detects that it is a pointer to a pointer to a function of a particular type. The necessary "marshall plan" (anonymous stub table) is generated automatically and associated with each pointer value generated by successive calls to `create`.

In fact, only one physical marshall plan is generated; it is reused with different "ministubs." A Concert/C ministub captures the specific information needed to find a particular server instance containing a remote procedure or queue. It arranges to call the right stub from the right marshall plan with the right routing information. This frees the application from any need to manipulate "RPC handles" or other explicit binding information and keeps RPCs "transparent" even while directing them to specific servers.

With any two-translators approach, since information from the two translators is merged only at link time, a static name (visible to the linker) *must* be assigned to each generated stub. If a stub is to be generated for each function *type* to support an indefinite (and dynamically determined) number of instances of that type, the add-on RPC package has *no choice* but to use a "client RPC handle" or similar construct to discriminate among server instances, making RPC inherently less transparent and requiring the application to become involved in the binding process.

3.3 Parallelism, Messaging, Passing Bindings

A further elaboration of our sample program shows some other convenience features of Concert/C. Suppose that the gathering of statistics at the server was expected to take a long time, so that it was useful to run all the servers in parallel. We would, therefore, like to restructure our application so that, first, RPCs are made to all servers asking them to start gathering the statistics, and then, as they gather them, the servers send the statistics back, asynchronously, in parallel. A set of declarations which is reasonable for this purpose is the following.

```
enum  status { success, not_authorized, system_error };
/* the above collapses what used to be 'status' and 'reason' */
struct report { /* report on one user at one machine */
   char * machine;
   char * name;
   unsigned long cpu;
   unsigned long memory;
   unsigned long disk;
};
/* the above is no longer a linked list, and a union is no longer used */
typedef receiveport {struct report} report_port;
typedef enum status start_gathering_t(report_port * answer);
```

Note that *no* Concert/C attribute lists are needed for this set of declarations. That is because no arrays (other than character strings), unions, aliased pointers, or parameters of uncertain directionality are employed. A new language extension is present, however, in the `receiveport` declaration. A `receiveport` is simply a queue of messages of any type. We also declare a function type designed to be the `initial` function of each server. This function is passed a `receiveport` pointer which will be kept by the server and used later to send results back. The initial function returns an `enum status` indicating whether the server is capable of carrying out the request.

The server would look something like this.

```
#include "example3.h"
```

```
report_port * answer_port;
initial port enum status start_gathering(report_port * answer)
{
 /* set up to gather statistics */
 if (/* can do it */)
   {
    answer_port = answer; /* save queue pointer */
    return success;
   }
 else
   /* return one of the failure codes */
}

main()
{
  struct report r;

  accept(start_gathering); /* one call to initial port */
  while (/* gathering statistics */)
    {
      /* gather record for a user in r */
      send(answer_port, r);
    }
  /* build special 'null user' record in r */
  send(answer_port, r); /* logical end of file */
}
```

The client to invoke this server would begin with the same logic as was used in the previous section, creating the servers and invoking their initial ports. However, as an argument to each call, it now passes the address of its `receiveport` called `answer_queue`. It would issue messages for those servers that couldn't perform the task, and count how many could. Then, it would do something like the following.

```
...
report_port answer_queue;
struct report r;
...
/* initial logic as described above */
...
while(server_count)
  {
   receive(answer_queue, &r);
   if (/* real user element */)
     add_to_report(r);
   else /* logical end of file for a server */
     server_count--;
  }
print_report();
}
```

Doing *something* similar in add-on RPC packages is sometimes possible, although many such packages do not provide asynchronous messaging. The most obvious way in which the single-translator helps here is that `ccc` can check whether the `send` and `receive` operations are type correct, finding errors sooner. In addition, the simple structure of the solution is greatly aided by being able to pass pointers to `receiveports` as first class values. These values are both *typed* and *anonymous*. The compiler generates the necessary "marshall plans" (anonymous stubs) automatically, and makes sure they are associated with each value of type "pointer to receiveport" as it is being received. First class typed, anonymous pointers cannot be provided with a "two translators" solution, again, because all stubs must be tied to a concrete symbol which is visible at link time. An add-on RPC package can, therefore, at best make some form of "connection handle" first-class. Although anonymous, such a handle is not typed, and therefore introduces more possibility of error.

4 Implementation

To enhance portability, the core of the Concert/C compiler is implemented as a preprocessor which executes after the ANSI C preprocessor, and before the ANSI C compiler. The compiler shell **ccc** invokes the local ANSI compiler in preprocess-only mode, passes the output through the Concert/C preprocessor, and feeds the result to the local ANSI compiler.

Before producing an executable, **ccc** invokes a pre-linker, which scans the object files and performs operations which require a global view of the program. This usually produces a small amount of code and data which is turned into an object file and combined with the other files to form the final executable.

To support debugging with plain C tools, and also to ensure that error messages from the C compiler are understandable, **ccc** passes ANSI C code directly into its output stream wherever possible, examining it without modifying it. Line number information is captured and preserved. Unlike the *cfront* implementation of **C++** [31], **ccc** does not "mangle" names. Concert/C programs, including ones instantiated by **create** can be debugged with tools such as **dbx** and **gdb**.

The compiler does, however, analyze all the declarations and statements in the program. Concert/C operations are checked and translated. The presence of any **port** function or **receiveport** declaration, or of an invocation of **create** (which may import one function or **receiveport** pointer), or a call to any function which has a function or **receiveport** pointer as an argument, triggers an analysis phase. The Concert/C type is examined and any missing attribute annotations are inferred and checked. These fully-annotated types are then fed into code which emits stub functions. Stubs are compiled for multiple protocols, with the final selection delayed until runtime.

The runtime is invoked on every RPC and every Concert/C operation. It dynamically selects among available protocols based on which pathways exist to the destination. Currently, it supports four protocols: SUN/RPC, OSF/DCE, our own UDP multicast protocol, and one based on shared memory "pointer passing" (between Concert/C processes implemented as threads in the same address space). The prototype permits Concert processes to interoperate with existing SUN/RPC components (such as NIS).

Many more details on the implementation are provided in [5].

5 Related Work

In the Concert project, we choose to extend multiple existing languages to best satisfy the conflicting goals of (1) hiding complexity and (2) building on an existing base of already-written code and programmer skills. A new language designed to support distributed computing (*e.g.*, Argus [20], SR [1], NIL [30], Emerald [10], Hermes [29]; a survey may be found in [7]) can hide complexity very well for those programmers willing to learn the new language. We will not engage in a feature by feature comparison with these languages, many of which have influenced Concert/C (particularly NIL and Hermes, which, like SR, share Concert/C's philosophy of providing both message-passing and RPC). Because such new languages must be learned, and programmers using them cannot easily draw on a large body of already-written code to perform routine tasks, they are not often used in the day-to-day practice of distributed computing. Also (although this is not inherent in the approach), the existing implementations of these new languages have not been "multi-protocol"; they are constructed in terms of a single protocol suite.

At the opposite extreme, commercial packages (such as OSF/DCE [25], Apollo NCS [19], PeerLogic's Pipes, Momentum's XIPC, Horizon's Message Express, or SUN ONC [32])) and software tools (such as Matchmaker [18], Courier [34], Horus [16], and HRPC [8]) permit programmers to continue using familiar languages and to incorporate existing code. Many of these packages are "multi-protocol" at the transport level (they work over many transport protocols), although only HRPC is "multi-protocol" at the RPC protocol level (interoperating with other RPC-based tools, as does Concert/C).

All of the tools in this second category are either pure library based or (more usually) follow the two-translators model (with a separate stub compiler). As we have attempted to show in our previous discussion of the Concert/C language, this approach limits the extent to which complexity can be hidden from the programmer. In addition, few of these tools provide for process management, often they provide only RPC or only messaging, instead of both, and the RPC-based ones often support demultiplexing of requests at the server only indirectly via an associated thread package. Partly because function is defined at a low level, but *mostly* because of the limitations of the "two translators" approach, the library APIs of the most powerful tools in this class become quite complex.[6] Tools attempting to address the residual complexity of library-based packages [24, 23] typically generate, in addition to stubs, a prologue to establish a program's initial connectivity, and also help automate the program construction process. Such tools are best at making simple "clients" that use a relatively static set of "servers." They provide only very limited help in writing servers, nor do they help when programs have evolving interrelationships.

The "single-translator" or "language extensions" approach of Concert/C is shared by some other efforts, most notably Linda [15]. Linda has a radically different model of computation than Concert/C (a shared tuple space), which makes it particularly effective for parallelizing single applications. However, we believe that Concert/C's model of computation, in which connectivity is modeled explicitly, is more appropriate for client/server applications (indeed, most of the tools cited above share Concert/C's view). In client/server applications, components often belong to semi-independent administrative domains, and the connectivity of the resulting "multi-application" is constantly changing.

Concurrent C [14] is also an extension to C, but we classify it as a new language because legacy C code almost always has to be reworked and must be recompiled in order to be incorporated into a Concurrent C program. Concurrent C has primitives for process creation and termination and has similar communication and synchronization semantics to Concert/C, but there are many differences reflecting Concurrent C's "single parallel application" design point. In Concurrent C, any process that has the other process' handle can invoke *any* of its exported procedures, and can also kill it; Concert/C provides a much finer granularity of control which is particularly important when crossing administrative boundaries. Concert/C provides all the necessary type system extensions to support the transmission of arbitrary C data structures across heterogeneous system boundaries; Concurrent C, lacking any such extension, requires either shared memory or a homogeneous distributed memory cluster. Concurrent C requires that the program bodies to be instantiated as processes be statically bound into a single program; Concert has no such requirement. One consequence of this difference is that Concert programs may be developed and instantiated by independent programmers in independent administrative domains and still communicate with each other.

PCN [12] is a new language with a C-compatible type system and the ability to link with C. However, when using PCN, all distributed logic is written strictly in PCN; C subroutines perform only local computation. With Concert/C, it is possible to make legacy C code directly invoke a remote function though originally written to perform only local function calls, or be invoked remotely though originally written to be invoked only locally. The degree of integration between old and new logic is therefore much greater with Concert/C.

As mentioned, while many tools support multiple transport protocols, almost none are designed to support multiple RPC protocols and also to interoperate with components using those protocols but not using the same programming tool. An exception is HRPC [8], a two-translators tool which does, however, have this multiple RPC protocols feature. 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

[6] For example, OSF/DCE [25] has a published programming interface comprising 157 RPC and threads function calls, not counting the ones used privately by stubs.

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.

6 Status

The current Concert/C prototype runs on AIX 3.2 on IBM Risc System/6000s, SunOS 4.1 and Solaris 2.2 on Sun workstations, and on OS/2 2.1. We have begun ports to VM/CMS and MVS, and are planning one to Windows/NT.

Programmers have used Concert/C to improve the performance of their applications in several ways, including parallelizing bottlenecks, and reducing communications by moving code next to its input data. For example, one large Concert/C application searches a database of genetic information on 64 Risc System/6000s in parallel [11]. Another Concert/C application uses similar search techniques to perform complex visual pattern recognition [27]. Concert/C has been used to connect a natural language textual query program to a collection of large, geographically dispersed text repositories [22]. The Global Desktop project uses Concert/C to enable the graphical interconnection of running applications, thus enabling cooperative computing over a network [21]. Yet another application has used Concert/C for the collection and analysis of time series data for the observation of seismic sensored phenomena [28].

The current prototype performs well. The round trip delay for an RPC that communicates a single integer takes 8 ms, little more than Sun RPC's 5 ms delay.[7] Transmission of arrays achieves a bandwidth of 0.5 Mbyte/sec. This performance compares favorably with other systems for parallel programming, like pvm [33] and C-linda [15], as studied in [13]. The measurement program is in the Concert/C Tutorial [17]. Parsons [26] has compared the usability and performance of Concert/C with other tools for distributed programming including Isis [9] and PVM [33].

7 Future Work

Concert/C is the first of a set of compatible extensions to important programming languages. Thus, we may build Concert/Cobol, Concert/Fortran, Concert/C++, etc. Each of these languages will implement a distributed computing model called the *process model*: a distributed program is composed of a set of sequential processes communicating by RPC and asynchronous message passing [35, 2]. These languages will interoperate, so that a Concert/Cobol process could serve an RPC by a Concert/C process. To support data interoperability, IPC messages are mapped in and out of a Concert Universal type family, as described in [6].

We are expanding the support for OSF/DCE to further exploit DCE services such as naming and security, and to provide richer interoperability with DCE components not written using Concert/C. These improvements will be made available in the anonymous ftp version later this year. We are currently designing Concert/C++, and are investigating preliminary designs for Concert/Fortran. We are also 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 the internal function signature representation as an intermediate form to mediate the search for equivalent representations in different languages. We are also investigating improvements to the run-time to add group communication and fault-tolerant primitives such as causal and atomic multicast [9].

The Unix implementations of Concert/C, together with comprehensive documentation, including a tutorial [17], a programmer's manual [4] and example code, are available via anonymous ftp [8].

[7] We measured this performance between IBM RS/6000 workstations connected by a 16Mbit token ring, and communicating via Sun RPC over TCP/IP.

[8] from `software.watson.ibm.com:/pub/concert`

References

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

[2] 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.

[3] 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.

[4] Joshua Auerbach, Arthur P. Goldberg, German Goldszmidt, Ajei Gopal, Mark T. Kennedy, and James R. Russell. Concert/C manual: A programmer's guide to a language for distributed C programming. Technical report, IBM T. J. Watson Research Center, 1993. To be published, available from the authors.

[5] Joshua S. Auerbach, Ajei S. Gopal, Mark T. Kennedy, and James R. Russell. Concert/C: Supporting distributed programming with language extensions and a portable multiprotocol runtime. Technical Report RC18995, IBM T. J. Watson Research Center, 1993.

[6] Joshua S. Auerbach and James R. Russell. The Concert Signature Representation: IDL as intermediate language. Technical Report RC19229, IBM T. J. Watson Research Center, 1993. To appear the 1994 ACM SIGPLAN Workshop on Interface Definition Languages.

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

[8] 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.

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

[10] 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.

[11] 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.

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

[13] C. C. Douglas, Timothy G. Mattson, and Martin H. Schultz. A comparison of distributed and shared virtual memory systems on networks. Technical report, Department of Computer Science, Yale University, New Haven, 1993. YALEU/DCS/TR-975.

[14] N. H. Gehani and W. D. Roome. *The Concurrent C Programming Language*. Silicon Press, 25 Beverly Road, Summit, NJ, 07901, 1989.

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

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

[17] 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.

[18] 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.

[19] 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.

[20] B. Liskov. Distributed programming in Argus. *Comm. ACM*, 31(3), March 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] Patricia Gomes Soares and Alan Randolph Karben. Implementing a Dlegation Model Design of an HPCC Application Using Concert/C. In *Proceedings of the 1993 IBM Centre for Advanced Studies Conference*, pages 729–738, Washington, DC, October 1993.

[29] 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.

[30] 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.

[31] Bjarne Stroustrup. A history of C++: 1979-1991. *SIGPLAN Notices*, 28(3):271–297, March 1993.

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

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

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

[35] 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.

Author Information

Joshua Auerbach is manager of Distributed Systems Software Technology at the IBM T J Watson Research Center, where he has been a Research Staff Member since 1983. His research has centered on problems of heterogeneity arising from system interconnection, including file systems, transport protocols, and now programming languages and RPC protocols. Email: jsa@watson.ibm.com

Arthur Goldberg is a technical assistant to the IBM Research Vice-President for Solutions, Applications and Services. He participated in the development of the distributed programming languages Concert/C and Hermes. He obtained his PhD in CS from UCLA in 91, and his AB in Astrophysics from Harvard in 77. Email: artg@watson.ibm.com

German Goldszmidt is a final-year PhD student in the Department of Computer Science at Columbia University. He has worked full-time and part-time for IBM Research for the past five years. His thesis research is in the area of network management. Email: german@cs.columbia.edu

Ajei Gopal received his PhD in Computer Science from Cornell University. He is a Research Staff Member at IBM Research, and manager of the Cluster Systems group. He previously worked in the Distributed Systems Software Technology group. His research interests center on parallel and distributed systems and fault-tolerance. Email: ajei@watson.ibm.com

Mark Kennedy received a BA in Chemical Physics and an MS in Computer Science from Columbia University. He has been a member of the Distributed Systems Software Technology group at IBM Research for the last four years. His research interests include distributed systems and doing anything to make it easier to program large collections of Unix-based workstations. He was a co-winner of the Winter '89 Usenix Humor Contest. Email: mtk@watson.ibm.com

Josyula R. Rao received his PhD in Computer Science from the University of Texas at Austin in 1992. He is currently a Research Staff Member at IBM Research in the Distributed Systems Software Technology group. His research interests include distributed systems and formal methods. Email: jrrao@watson.ibm.com

James Russell received his PhD in Computer Science from Cornell University. He is a Research Staff Member at IBM Research, where he has been working in the Distributed Systems Software Technology group for the past three years. His research interests focus on the design and implementation of programming languages for distributed applications. Email: jrussell@watson.ibm.com