

High-Level Language Support for Programming Distributed Systems

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 S. A. Yemini

IBM T. J. Watson Research Center
Box 704, Yorktown Heights, NY 10598
yemini@watson.ibm.com (914) 784-7627

Abstract

This paper presents a strategy to simplify the programming of *heterogeneous distributed* systems. Our approach is based on integrating a high-level distributed programming model, called the *process model*, directly into programming languages. Distributed applications written in such languages are portable across different environments, are shorter, and are simpler to develop than similar applications developed using conventional approaches.

In this paper, we discuss the process model, and present overviews of *Hermes* and *Concert/C*, two languages that implement this model. *Hermes* is a secure, representation-independent language designed explicitly around the process model. *Concert/C* is the C language augmented with a small set of extensions to support the process model while allowing reuse of existing C code. *Hermes* has been prototyped; an implementation of *Concert/C* is in development.

1 Introduction

With the advent of distributed computing technology, there has been an increased interest in *multiapplications*—systems consisting of multiple interacting applications, running in a heterogeneous network. Developing multiapplications using conventional technology is not easy, for the following reasons.

Conventional programming languages reflect the assumptions of an earlier generation of computing in which applications do not interact with each other. To enable an application to communicate with another, the programmer must “step outside” the language and use operating system services (such as sockets in Unix) or various remote procedure call (RPC) services. Thus, the programmer of a multiapplication

must master several distinct interfaces: those provided by the programming language, the operating system, and “add-on” packages.

To further complicate matters, multiapplications may be developed from applications that are written in different languages, assuming different operating systems and different network environments. Thus, programmers of multiapplications must master *multiple* languages, type systems, operating systems and network environments, consider their interactions, and provide mappings between them. This often forces the programmer to deal with the lowest common denominator of the implementations—the bits and the message protocols.

Our goal is to make it easy for a programmer to develop a system consisting of interacting applications, written in diverse languages and executing on diverse machines, without any knowledge of the details of the underlying operating system, communications protocols or hardware. To this end, we define an abstract computation model supporting process creation, process interconnection, and inter-process interaction. This model leads naturally to a family of interoperating languages, including new languages and enhanced versions of existing languages, that provides a flexible yet simple framework to develop multiapplications.

Before introducing our approach (Section 1.2), we describe more conventional approaches taken to support the development of distributed applications.

1.1 Conventional Solutions

Two approaches are commonly used to simplify the task of developing distributed applications: the use of the remote procedure call (RPC), and the use of new programming languages.

1.1.1 The Remote Procedure Call (RPC)

Informally, an RPC facility enables a call in one address space to invoke a procedure within another ad-

dress space, possibly on a remote machine [1]. Most RPC packages are designed to hide some of the details of the underlying transport protocols and automatically handle much of the problem of converting data to and from transmittable form [2, 3, 4].

Unfortunately, even with today's RPC packages, developing distributed applications remains a task for an expert. This is because the programmer must master a large and complex programming interface, one that can be substantially different from the familiar one presented by the programming language.

In most systems, RPC services are offered by an add-on package, that typically includes an "interface definition language," a "stub generator," and a very large number of library functions, that are often at too low a level of abstraction. For example, before a remote call is made, the callee must register with the RPC runtime, network services, name services, *etc.*, and must then listen on a socket for arriving calls. The caller must invoke network services, name services, binding services, and other RPC services to locate the remote procedure and establish a remote binding.

With conventional RPC systems, the semantics of a remote call is often different from that of a local call. For example, such systems do not allow procedure pointers to be passed as parameters, even in languages where this is allowed in the local case.

Finally, conventional RPC services do not permit interoperability; that is, a server written assuming one particular RPC package cannot in general be invoked by a client written assuming another RPC package.

1.1.2 Distributed Programming Languages

Programming languages that were designed to support distributed applications include object-oriented languages such as Emerald [5] and functional languages such as Concurrent ML [6] (see [7, 8] for discussions of such languages). A comparison of such programming languages with our approach is omitted from this preliminary version of the paper.

1.2 Our Solution

There are three key aspects to our approach to simplifying the development of multiapplications: the *process model*, an integration of programming language and system interfaces, and interoperability.

Process Model: The process model defines a common language-independent and system-independent computation model supporting process creation, process interconnection, and inter-process interaction. Thus,

a small number of common abstractions in the model replace the hundreds of interfaces in today's library packages for distributed computing. In Section 2, we describe the process model and show how it can form the basis for programming heterogeneous distributed multiapplications.

Integrated Language and System: We integrate the programming language and system by supporting the process model directly within new languages, and by minimal, seamless extensions to existing languages. Thus, we eliminate the multiple abstractions of language, operating system and add-on packages, and replace them with a single, higher-level abstraction. In such an integrated system, the programmer uses a single programming interface, ignoring the details of how this interface is supported. It is the responsibility of the implementer of the language (and not of the user), to transparently map this interface into low-level services provided by the language run-time system or by the underlying operating system.

In Section 3, we describe *Hermes*, a new language we have developed which fully incorporates the process model. In Section 4, we describe *Concert/C*, a set of extensions to C which form a second fully compatible embodiment of the process model.

Interoperability: In general, other distributed programming languages do not support interoperability with other languages, or with existing code. In contrast, programs written in our family of languages interoperate with each other and also with servers written using conventional RPC packages. This is done in the language run-time system, and is transparent to the programmer (Section 2.1.3).

2 The Process Model

At the heart of our framework for programming distributed systems is the simple yet powerful computational model we have referred to as the *process model*. The process model specifies how processes are created and interconnected, and how they interact.

2.1 An Overview of the Model

The main components of the model are:

- The concepts: process, port, and binding.
- The set of language-independent data types which can be communicated between processes.
- An interface definition mechanism for specifying *contracts* which communicating processes agree to follow.

- Operations for creating and destroying processes, creating and destroying bindings, and communicating between processes.

2.1.1 Processes, Ports, and Bindings

Each component of a multiapplication is a process. A *process* is an active entity that performs computations. Each process contains: (1) a program, and (2) local data variables, consisting of computational variables and *ports*. A process may be written in any language, and may do any of the following:

- Computation — reading, writing, and updating computational variables.
- Communication — sending and receiving data over a port.
- Creation — making a new process by instantiating a program.
- Connection — making a new binding by connecting an output port to an input port and sending one of the ports away.

A port is a communications endpoint. Ports are either *input ports* (abbreviated as *inports*), or *output ports* (*outports*). An inport is a queue, holding data which has been sent but not yet received. An outport holds a *binding* to an inport — that is, the right or capability to send data to that inport. Several outports may contain a binding to the same inport. Messages sent using any given outport arrive at the associated inport in the order sent; messages from multiple outports bound to the same inport are merged fairly.

2.1.2 The Type System

The process model includes a universal language-independent type system defining the datatypes which may appear in messages exchanged between processes. The type system includes the following types:

- Traditional scalar data: boolean, integer, real
- Enumeration: a value from a particular finite domain
- Port: inport or outport adhering to a particular contract
- Program: code capable of being instantiated
- Process-id: the capability to cancel a particular process
- Nominal: an identifier capable of serving as a *label* or a *reference*, useful for building complex data structures such as graphs.

- Composites:

- Record: an indexed tuple of values — each component is of a specified type
- Callmessage: a record containing *in-parameters* together with an outport (the *return* outport), over which a reply containing *out-parameters* is to be sent. (In Hermes and Concert/C, the return outport is not explicitly accessible to the programmer.)
- Choice: a value which may have one of a finite set of possible types, plus a *discriminant* defining the actual type
- Polymorph: a value which may be of any type, plus a *descriptor* defining the actual type.
- Bag: an indefinite-sized collection of values all of a particular type
- Array: an indefinite-sized ordered bag

Notice that port values are first-class data and may themselves be sent in messages. This permits a great deal of flexibility for dynamic configuration of processes, and for “plug replacement” of one service by a different service with the same interface. For example, any user may program a specialized “directory service” with his own policy for returning bound outports to clients. The user may select which outport to return based upon time of day, load, etc.

2.1.3 Contracts

An output port may be bound to an input port only if the two ports agree to the same *contract*. Informally, a contract defines a set of “promises” by the sender and assumptions by the receiver. The simplest contract specifies the types of data being sent in a message: for example, the sender may promise to send a record containing an integer and an array of characters.

More complex contracts involve *constraints*. Constraints may either relate the values sent in a message, or relate in-parameters of a callmessage to the out-parameters. Here is an example of a contract which does both: The sender sends a callmessage containing an integer I, an array of reals A, and an outport for returning an array of reals B, subject to the constraint that I is equal to the size of A, and the size of A is equal to the size of B.

We have developed a language-independent notation for writing contracts. However, each language supporting the process model has the option of providing its own syntax for expressing contracts. For example in Concert/C, function declarations augmented by annotations simultaneously specify the contract and the Concert/C data type, eliminating the requirement

for programmers to write separate files in a separate “contract” syntax.

Having a language-independent type system and formal contract semantics supports heterogeneous computing. Each communications endpoint is free (subject to the constraints of the language, hardware, and runtime environment at that endpoint) to choose any representation which satisfies the semantics of the contractual type. For example, what is described as an array of records in the contract may be represented in one process as a linked list of records, and in another as a zero-terminated array of pointers to records. If two communicating processes use different representations, it is up to the runtime environment to either convert between the representations, or for the sender to “marshall” data into a standard format and for the receiver to “demarshall” from that format.

2.1.4 Universal Operations

Any language supporting the process model will support the following operations:

- **send**(message, outport): Enqueue the message on the inport to which the outport is bound. The sending process does not wait for a reply.
- **receive**(message, inport): Dequeue a message from an inport, or wait if no message is currently available.
- **select**(inport-list): Wait, if necessary, until one of the inports has a message. If one or more inports has a message, return the index of one of these inports.
- **new**(inport) Initialize the inport with an empty queue.
- **connect**(inport, outport): Initialize the outport with a binding to the specified inport.
- **create**(program, process-id, outport): Create a new process running the designated program. Initialize the process-id variable to the process-id of the new process. Initialize the outport to a binding to a designated *initialization inport* in the newly created process. This binding serves as an “umbilical cord” through which the parent process and the child can exchange parameters.

Although calls are not semantically primitive, since they reduce to bi-directional sends and receives, they are such a widespread programming idiom that they are included here in the list of universal operations:

- **call**(outport, in-parameters, out-parameters): A callmessage is created, containing the in-parameters and a binding to a *reply inport*. The callmessage is then sent over the outport. When a reply arrives on the reply inport, it is dequeued, and the data is moved from the callmessage to the out-parameter variables. A consequence of the callmessage’s being a first-class data value is that the called process may forward it to a third party and the caller may receive the reply from that third party.

- **reply**(callmessage, out-parameters): The out-parameters are sent over the return port in the callmessage. (In Hermes and Concert/C, space is reserved in the callmessage to hold the out-parameters.)

An attractive feature of the process model is the fact that a small number of relatively simple operations provides the same expressive power as a larger number of lower-level, platform-specific operations offered by operating system or RPC environments.

2.2 Implementations

Because the universal types and universal operations are defined abstractly, there is a wide latitude for different implementations. Not only may processes be implemented in different ways, but also multiple alternative implementations may coexist and interoperate in the network. We will briefly discuss two additional dimensions along which implementations will differ: (1) exclusivity and enforcement, and (2) efficiency.

2.2.1 Exclusivity and Enforcement

The process model includes no mechanism for two processes to share data, for a process to access non-local data other than through ports, or for a process to obtain a port to a process other than by being that process’s parent or by receiving that port through a chain of communication originating from that parent.

An implementation is *exclusive* if it forbids the programmer to do anything not explicitly allowed by the process model; otherwise it is *non-exclusive*. For example, a non-exclusive environment might allow a process to communicate directly with the operating system of the machine on which the process runs, without going through ports. We expect to provide both exclusive and non-exclusive implementations; each has particular advantages.

Exclusive environments make it simpler to reason about the behavior of processes: processes do not interfere with one another; the behavior of a process

does not depend on where it is located. Exclusive environments also provide an element of access control, since one can rely on the unforgeability of ports. On the other hand, non-exclusive environments allow a vast amount of existing code to interoperate in a distributed environment without being rewritten.

If an implementation does forbid certain behavior, there is also a spectrum of enforcement: (1) no enforcement — the system may “crash” if the rule is violated, (2) violations of a rule will be detected either at compile-time or run-time, (3) the implementation may guarantee proper behavior of the system even in the presence of machine failures (see in [9]).

2.2.2 Efficiency

There are many implementation decisions affecting efficiency: e.g. data structure selection, assignment of processes to processors, sharing of many small processes within a single address space, etc. Additionally, implementations are free to perform arbitrarily aggressive transformations of programs, provided the semantics of the process model is preserved.

Examples include: migrating processes[10], replicating processes [11], converting bi-directional calls to sends[12].

3 Hermes

Hermes [13] is a new experimental high-level language especially suited for distributed and multiapplication systems. It is a very high level language that fully incorporates the process model. In particular,

1. The concepts of process, inport, outport, etc. are primitive concepts of Hermes.
2. Hermes enforces an exclusive implementation of the process model by compile-time checking.
3. Processes are the fundamental units of modularity. Procedures, “objects”, “servers”, and “filters” are all processes.
4. Hermes hides data representations from the programmer.

Hermes supports the complete set of process model operations defined in the preceding section.

3.1 Novel Features

In many ways, Hermes is a conventional algorithmic language in the Algol tradition. Each process

executes a sequential imperative program consisting of statements operating on variables. Hermes differs from conventional languages in the following ways:

- *“Featherweight” Processes:* Hermes implementations support systems containing many hundreds or thousands of processes. A call from one process to another is expected to be as efficient as a procedure call in a conventional language.

- *Typestate:* In Hermes, the legality of applying operations are formalized as *typestate rules*, so called because the legality of an operation depends on the type and *state* of the operands. Programs are checked for conformity to the typestate rules using a dataflow algorithm [14]. All operations in a compiled Hermes process conform to the typestate rules—in particular, uninitialized data and memory belonging to another process cannot be accessed. As a result, any number of Hermes processes can safely share a single address space. A bug in one of these processes—e.g. an uninitialized variable error—cannot have the side effect of bringing down other processes.

- *Representation-independent Data Types:* A Hermes programmer defines a data type without describing its underlying machine representation. Hermes stores aggregate data in *tables*, which are similar to database relations. A table’s type is defined by describing the data type and typestate of its elements. Tables are first class objects: a table can store a table in one of its elements; a table can be passed in messages. A table can be expanded and contracted using *insert* and *remove* statements.

The physical representation of a table in memory is determined by the compiler, using advice given by the programmer in the form of *pragmas*. There are no operations (such as C’s *sizeof*) which permit access to physical properties of the representation.

- *First class typed ports:* Unlike many object-oriented systems, where the object interface (class description) describes not only the contractual properties of the object interface but also its implementation (e.g., methods), the type of a port defines only what is needed to interoperate with the process owning the port. Different processes are free to implement any code they desire, as long as the contract is obeyed.

- *Dynamic Program Creation:* In Hermes, programs are first-class objects that can be dynamically constructed *and checked* at run-time and then executed. (Typestate checking guarantees that only checked programs are instantiated as processes).

3.2 Examples

The following example shows a Hermes system in which a server process stores a table of programs, and another process retrieves a program from the server and then instantiates the program.

We begin with a Hermes type definition for the table which stores programs:

```
Labeled_Program: record(  
  Label: Charstring,  
  Prog: Program);  
Repository: table of Labeled_Program  
  {init(Label), init(Prog), checked(Prog)}  
  keys(Label);
```

The type `Labeled_Program` defines a pair consisting of a character string and a program. `Repository` defines a type whose values are arbitrary sized sets of `Labeled_Program` pairs. The information in curly braces indicates the typestate to be enforced for each element of the table—in this case, both components of the pair must be initialized and the program stored in the `Prog` field must be checked. The `keys(Label)` specification indicates that no two elements of the table can have the same value of `Label`.

The interface for the operation of getting a program from the repository includes three type definitions—those for its callmessage, inport and outport.

```
Get_Prog_Inter: callmessage (  
  Name : charstring,  
  Prog : program)  
  constant(Name)  
  exit{ init(Name), init(Prog), checked(Prog) };  
  exception NoSuchProg { init(Name) };
```

```
Get_Prog_In: inport of Get_Prog_Inter  
  { init(Name) };
```

```
Get_Prog_Out: outport of Get_Prog_In;
```

The callmessage `Get_Prog_Inter` contains two parameters: an in-parameter (`Name`) and an out-parameter (`Prog`). In-parameters and out-parameters are distinguished and further constrained by means of typestate annotations on the inport and callmessage definitions. In this example, `Name` is initialized on input and remains initialized on exit. `Prog` is both initialized and checked on normal exit, but uninitialized upon the exceptional exit `NoSuchProg`. The declaration constant states that `Name` is not changed by this program and is therefore not an out-parameter. `Get_Prog_Out` is an outport type which can be connected to `Get_Prog_In` typed inports.

This interface is used by a server process that stores a table of programs:

```
program_repository : process ( ... )  
  
declare  
  Get_Prog: Get_Prog_In;  
  Get_Prog_CM: Get_Prog_Inter;  
  R: Repository;  
  
begin  
  -- Initialize R, Get_Prog, ...  
  ...  
  while ('true') repeat  
    select  
      event Get_Prog  
        receive Get_Prog_CM from Get_Prog;  
  
      block  
        begin  
          inspect Entry in R where  
            (Entry.Label = Get_Prog_CM.Name)  
          begin  
            Get_Prog_CM.Prog <- Entry.Prog;  
          end inspect;  
          return Get_Prog_CM;  
        on(NotFound)  
          return Get_Prog_CM exception NoSuchProg;  
        end block;  
  
      event Put_Prog  
        -- code to insert and delete programs  
        ...  
    end select;  
  end while;  
end process
```

When this server is instantiated it initializes an empty repository and creates inports for the services it exports: get a program, insert a program, etc. (this code is not shown). In the infinite loop this process repeatedly waits for a callmessage on some input port, receives the callmessage, takes the appropriate action and then returns the callmessage. The `inspect` statement locates a labeled program with the the right name and stores it in the `Prog` field of the callmessage. The `return` statement performs the reply operation of the process model, using the `Prog` field of the callmessage as the out-parameter. (If the requested name was not found in `R` then the `NotFound` exception handler would execute, which returns the callmessage with the appropriate exception.)

The following code obtains a program labeled "foo" from the repository server and instantiates it:

```

declare
  Get_Prog_Fn : Get_Prog_Out;
  P : Program;
  Init : Some_Outport_Type;
  ...
begin
  ... -- initialize Get_Prog_Fn
  call Get_Prog_Fn( "foo", P );
  Init <- create of P;
  call Init( ... );
  ...

```

The first statement gets a program named "foo" from the repository and stores the program in P. The second statement starts a new instance of the program P. Init contains an "umbilical cord" binding to the initialization input port of the child process. The third statement calls the initialization port of the child process.

The following fragment is illegal and will be rejected by the compiler:

```

declare
  Get_Prog_Fn : Get_Prog_Out;
  P : Program;
  Init : Some_Outport_Type;
  ...
begin
  -- initialize Get_Prog_Fn
  ...
  Init <- create of P;
  call Init( ... );

```

In this example we have removed the call which initializes P. The typestate preconditions for a create operation requires that its argument be initialized. Since P wouldn't be, this will be caught as a typestate error. If such an error were not caught the program would point an outport to an uninitialized program. When a call was made later on the outport it could potentially crash the system. Hermes' typestate checking has the effect of catching a bug early, and avoiding a crash.

3.3 Summary

Hermes is a new language which fully realizes the process model and fully integrates itself with the system. A programmer can create numerous small processes, each as small as a single module, which would not be practical in other languages. Hermes' extensive static checking catches a larger number of programming errors at compile-time. Additionally, modules which compile successfully can safely run together in

an address space with other Hermes modules, resulting in a finer granularity of security at lower cost. Finally, the higher-level, representation-independent data types simplify coding, enhance portability, and enable an open-ended set of optimizations.

4 Concert/C

Despite the many advantages of Hermes, it is a new language, and organizations and programmers have a considerable investment in existing languages. The Concert project aims to incorporate the process model into the most widely used programming languages, making it possible to build on existing code repositories and programmer skills. These extended languages interoperate with Hermes as well as with each other: thus the strengths of Hermes and these extended languages complement each other.

We now discuss how the C language was extended to form Concert/C. We show how each of the key entities and operations of the process model was added to C by generalizing an existing C concept, or adding a new type or operation.

4.1 Processes

A process has its own thread of control and its own data. In Concert/C, a process corresponds to an instantiated and executing C program. The code of the process is the code of the main function and all functions statically linked with it. The data of the process is the contents of the static (file-level scope), automatic (block-level scope), and dynamic (heap-allocated) storage accessible from the process.

C has no notion of processes, and hence no types or primitives for describing or instantiating them. Therefore, Concert/C augments C with two new types, `prog_t` and `proc_t`, and a new statement `create`.

A `prog_t` corresponds to the program type of the process model – it is a process description that can be instantiated to a running process. A `proc_t` is a handle on a running process, providing the ability to terminate it.

`create` is the operation that takes a `prog_t` and instantiates it into a running process. It returns a `proc_t` handle, and sets up a binding to the initial port within the process.

4.2 Ports

In Concert/C one declares an inport by giving the `port` storage class attribute to a function. The mes-

sage type of the port is a `callmessage` whose parameters are derived from the signature of the function. The function signature may include Concert/C *attributes* which augment and disambiguate the C types in order to define a contract. The new type specifier `receiveport{ <msg-type> }` defines an inport for a type `msg-type` other than a `callmessage`. Receiveports always have the `port` storage class.

The `accept` operation (described below) combines dequeuing a message from an inport with invoking the associated function. The `receive` operation dequeues a message without invoking the function.

An outport is represented as a pointer to a member of the port storage class. Such pointers are first-class objects and can be copied or passed to remote processes. A pointer to a remote queued function can be viewed as a *closure* – it identifies both a code body (the function which will be invoked) and the context in which the code body executes (the data accessible to the invoked function within the process). The calling semantics of these function pointers are described below.

4.3 Communication

An inter-process call within Concert/C has the same syntax (and semantics) as an ordinary C function call. Invoking a function pointer whose referent is local performs the standard C function call. Invoking a function pointer whose referent is non-local transparently performs a process model call: a `callmessage` is created and queued at the remote queue; the caller blocks until the call returns.

To support the callee side of a process model call, Concert/C adds three new operations, `receive`, `reply`, and `accept`, and one new data type `msg`. The process receiving a remote call dequeues the call by performing the `receive` operation on (a pointer to) the corresponding port function. The result is an object of type `msg`, which corresponds to the `callmessage` of the process model. The fields of the `msg` are the formal parameters of the function, which is then invoked by the process. When the function returns, the process issues a `reply` on the `msg`, which returns the `callmessage` and unblocks the caller.

The `accept` operation of Concert/C simplifies the above tasks. It takes an arbitrary number of function pointer arguments, blocks until one of the queues receives a `callmessage`, and automatically performs the above series of actions (`receive`, function invocation, and `reply`).

Concert/C also supports the `send` operation of the process model. The construct `send (port_ptr,`

`data)` performs a one-way send of data to the `receiveport` pointed to by `port_ptr`. `Send` does not block, and the type of data must be the same as the `receiveports` `msg-type`. The `receive` operation is used on a `receiveport` pointer to dequeue the data.

To support the queue management functions of the process model, Concert/C adds the operations `poll` (test whether a message is available on a port) and `select` (wait for a message to be available on one of a set of ports).

4.4 Program Profiles

Concert/C processes can be connected to one another either statically or dynamically. As mentioned above, dynamic binding is achieved by passing, storing, and retrieving function pointers. Static (i.e. load-time) binding is achieved using *program profiles*.

A program profile indicates those functions within a program which need to be exported, or imported, and their locations or destinations. A program profile can also indicate more complex relations for composing distributed applications, such as generating prototypical server code or building a process “graph” by having one process start others. The Concert/C compiler then generates appropriate *initialization stub* for the program from the program profile. This stub hides all the runtime calls to distributed computing services needed to initialize the bindings.

4.5 An Example

Simple clients, programs which make remote calls but do not receive them, can be written using only a few Concert/C extensions. Consider the following Concert/C statements:

```
typedef int client_request_t(
    int account,
    int amount,
    [out] int *status);
client_request_t *withdraw, *deposit;
client_request_t *(*find_atm)(
    [in, string] char *location,
    [in, string] char *operation);
```

First, the type `client_request_t` is defined. It describes the function signature of a deposit or withdrawal request. Next, the function pointers `withdraw`, and `deposit` are declared. These will eventually be initialized with pointers to remote queued functions. Finally, the function pointer `find_atm` is declared. It points to a function which, given the location of a bank client and the name of a client operation, will return

a function pointer of type `client_request_t` to perform the operation named. Calls to this function will be used to initialize `withdraw` and `deposit`.

Concert/C annotations to the function parameter declarations appear in the example. These are comma-separated lists of keywords within square brackets. The style of annotation (and many of the attribute names) are based on the Interface Definition Language of the Open Software Foundation's Distributed Computing Environment (OSF DCE).

Now we consider the sample program proper.

```
[[ dirarch osf_dce_gds; import find_atm { from
    "/Country=US/Organization=Infidelity Trust"
    "/OU=Customer Services" }; ]]
```

```
void main(void)
{
    int allowed, my_balance;
    int my_account = 12345678;
    int my_lottery_check = 10000;

    deposit = find_atm("My Net", "Deposits");
    withdraw = find_atm("My Net", "Withdrawals");
    deposit(my_account,
            my_lottery_check, &allowed);
    my_balance = withdraw(my_account,
                        200, &allowed);
    if (! allowed)
        printf("Time to buy another ticket?\n");
}
```

The only Concert enhancements needed in the entire example are the *program profile* statements bracketed by `[[` and `]]`. They resolve the external reference to the function pointer `find_atm` using a value from outside of the current process. In this case, the pointer is imported from an X.500 directory service. We assume the bank has exported it into the same directory service at an earlier time. The purpose of the program profile was discussed in section 4.4.

Otherwise, the application is straightforward. The queued function to which `find_atm` points is invoked twice in the usual way, resulting, however, in procedure calls to a remote process. That process, in turn, returns pointers to two more remote queued functions (the desired client operations). The main body of the program eventually calls these functions and these, too, result in remote calls.

Now let's consider a server to process the requests of our simple client program. Here is the body of the server:

```
port client_request_t withdraw, deposit;
```

```
[[ dirarch osf_dce_gds;
    export withdraw, deposit { to
        "/Country=US/Organization=Infidelity Trust"
        "/OU=Operations/Branch=N"; }
    automain;
]]

int withdraw(int account_number,
            int amount, int *status)
{ /* perform the debit and record it */ }
int deposit(int account_number,
            int amount, int *status)
{ /* perform the credit and record it */ }
```

The functions `withdraw` and `deposit` are declared to be in the port storage class, and thus queues will be allocated for each. The program profile exports these two functions using appropriate names. Another server will retrieve these functions from the directory response to the `find_atm` requests of clients.

The `automain` statement requests that the C main function be generated automatically by Concert/C. The generated main program will simply accept calls to any functions exported in the program profile (in this case, `withdraw` and `deposit`) until the server fails or is cancelled by its parent. Only one call is processed at a time, so the server doesn't have to worry about concurrent access to the account data. Greater concurrency, if desired, can be achieved by other means within Concert/C.

Now suppose that greater control is needed over server behavior. We would like to have reserved forms of `withdraw` and `deposit` to be used only by authorized personnel, for example to correct any errors in the ATM's account information. We can do this by defining two new `client_request_t` functions, `reserved_withdraw` and `reserved_deposit` which will be passed only to processes in the bank's Operations division. We will need to write explicit accept statements in the server's main program instead of generating it automatically. And, we will define two control operations, so that while corrections are being made, regular customer requests can be deferred. The customer requests are enqueued, rather than discarded.

The body of the new server appears below:

```
port client_request_t withdraw, deposit;
port client_request_t reserved_withdraw,
                    reserved_deposit;
typedef void sync_request_t(void);
port sync_request_t reserve, relinquish;
```

```
[[ dirarch osf_dce_gds;
    export withdraw, deposit,
```

```

    reserved_withdraw, reserved_deposit,
    reserve, relinquish { to
    "/Country=US/Organization=Infidelity Trust"
    "/OU=Operations/Branch=N"; }
]]

int publically_available = TRUE;

void main(void)
{
    for (;;)
        if (publically_available)
            accept(reserve, withdraw, deposit);
        else
            accept(relinquish, reserved_withdraw,
                reserved_deposit);
}

/* Define withdraw and deposit functions just as
   in the first server example. In addition: */
void reserve(void)
{ publically_available = FALSE; }
void relinquish(void)
{ publically_available = TRUE; }
int reserved_withdraw(int account_number,
    int amount, int *status)
{ return(withdraw(account_number,
    amount, status)); }
int reserved_deposit(int account_number,
    int amount, int *status)
{ return(deposit(account_number,
    amount, status)); }

```

The body of `main` is simple: an infinite loop which conditionally accepts requests from the public (`withdraw`, `deposit`) or from the Operations division (`reserved_withdraw`, `reserved_deposit`) depending on the value of the global, static variable `publically_available`.

The new type `sync_request_t` is used by both the control functions `reserve` and `relinquish`. Since only one call is processed by each `accept` statement, nothing need be done to prevent or mediate concurrent access to the variable `publically_available`.

The bodies of the *reserved* forms of `withdraw` and `deposit` simply call their normal, un-reserved counterparts. There is nothing special about calls to `withdraw` and `deposit` from within the server process. They are *normal* C function calls. The reserved function definitions are exploited only for their ability to define additional queues, distinct from the ones associated with `withdraw` and `deposit`.

5 Status and Conclusions

5.1 Hermes

IBM Research is distributing, without charge, an experimental *prototype* implementation of Hermes to anyone who wants to use it for non-commercial purposes. The implementation currently runs on many Unix platforms, including the IBM RISC System/6000 running AIX or Mach 2.5, the IBM RT/PC running 4.3 BSD Unix, Sun 3's and SPARCstations running SunOS 4.0.3 or higher, HP 9000s running HP-UX, the NeXTStation, the DEC MIPS running Ultrix and the Convex.¹ Hermes is implemented as a run-time written in C and a compiler written in C and Hermes. It should port easily to most 32-bit systems. We distribute all source for the compiler and run-time.² We have published a tutorial and reference manual [13].

The Hermes run-time environment supports a single logical Hermes machine distributed over a network that can include any machine Hermes runs on. Hermes compiles to a combination of C modules and a Hermes intermediate language.

Hermes is a descendent of an earlier language, NIL. The NIL implementation ran only on VM/370 and was highly optimized. For example, making calls between processes took 9 machine instructions, far less than in any lightweight process system. NIL was evaluated by using it to prototype two large SNA subsystems. This experience was encouraging both from the standpoint of software engineering and from the standpoint of performance.

5.2 Concert

The existing Concert prototypes [15] demonstrate processes written in C running under AIX or OS/2 on PS/2 hardware calling each other and also VM/370 processes written in PL/I. Supported communications protocols include 3270 terminal emulation, APPC, NETBIOS, and TCP/IP. To bridge the heterogeneity of these environments, a communications package called TACT[16] is used.

We are now engaged in designing the second-generation Concert prototype including the features omitted from the first prototype. "Concert II" will run on multiple system platforms, and will be distributed

¹Trademark or registered trademark of UNIX System Laboratories, Inc., International Business Machines Corporation, Sun Computer Corp., Hewlett-Packard, Digital Equipment Corp. and Convex Corp.

²Hermes can be anonymously ftp'ed from IBM Research's ftp site, software.watson.ibm.com.

as experimental software on the same basis as Hermes. We have temporarily deemphasized other potential Concert languages in favor of finalizing a complete and rigorous definition of the Concert/C language. A paper giving a complete description of this language is currently available[17].

5.3 Conclusions

The process model appears to be a simple yet expressive base for supporting distributed computing across diverse languages and system environments. Hermes and its predecessor NIL have shown that high level languages can significantly simplify programming complex systems and yet still have highly efficient implementations. With Concert/C we have shown that conventional high level languages can be extended to hide the complexity of distributed computing with a minimum of change to the language or to existing code. We plan on integrating Hermes and Concert/C into a single environment.

References

- [1] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, pp. 39–59, February 1984.
- [2] Sun Microsystems, *SUN Network Programming*, 1988.
- [3] M. Kong, T. H. Dineen, P. J. Leach, E. A. Martin, N. W. Mishkin, J. N. Pato, and G. L. Wyant, *Network Computing System Reference Manual*. Englewood Cliffs, NJ: Prentice-Hall, 1990.
- [4] Open Software Foundation, *OSF DCE Release 1.0 Developer's Kit Documentation Set*, February 1991.
- [5] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and abstract types in emerald," *IEEE Transactions on Software Engineering*, vol. 13, pp. 65–76, January 1987.
- [6] J. H. Reppy, "CML: A higher-order concurrent language," in *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pp. 293–305, ACM, June 1991.
- [7] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum, "Programming languages for distributed computing systems," *ACM Computing Surveys*, vol. 21, September 1991.
- [8] G. A. Andrews, *Concurrent Programming: Principles and Practice*. Benjamin/Cummings Publishing Company, Inc., 1991.
- [9] R. E. Strom and S. A. Yemini, "Optimistic recovery in distributed systems," *ACM Transactions on Computer Systems*, vol. 3, pp. 204–226, August 1985.
- [10] W. Korfhage and R. Yozzo, "Picking processes for migration," in *Proc. 4th ISMM/IASTED Intl. Conf. on Parallel and Distributed Computing Systems*, pp. 193–195, Oct 1991.
- [11] A. P. Goldberg, *Optimistic Algorithms for Distributed Transparent Process Replication*. PhD thesis, University of California at Los Angeles, 1991. (UCLA Tech. Report CSD-910050).
- [12] D. F. Bacon and R. E. Strom, "Optimistic parallelization of communicating sequential processes," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, April 1991.
- [13] R. E. Strom, D. F. Bacon, A. Goldberg, A. Lowry, D. Yellin, and S. A. Yemini, *Hermes: A Language for Distributed Computing*. Prentice Hall, January 1991.
- [14] R. E. Strom and S. A. Yemini, "Typestate: A programming language concept for enhancing software reliability," *IEEE Transactions on Software Engineering*, vol. SE-12, pp. 157–171, January 1986.
- [15] 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*, pp. 162–171, IEEE Computer Society, June 1989.
- [16] J. Auerbach, "TACT: A protocol conversion toolkit," *IEEE Journal on Selected Areas in Communications*, vol. 8, pp. 143–159, January 1990.
- [17] J. Auerbach, M. Kennedy, J. Russell, and S. Yemini, "Interprocess communication in concert/C," Tech. Rep. RC17341, IBM T. J. Watson Research Center, October 1991.