Hermes: An Integrated Language and System for Distributed Programming

Robert E. Strom
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, N.Y. 10598

Abstract

Hermes is an experimental language for implementing complex systems and distributed applications. It conceals low-level programming details, such as data representation, distribution, communications protocols, and operating system calls, while retaining expressiveness, checkability, and efficiency. Hermes is being distributed free of charge to the research community.

Hermes supports multiple interacting applications and services within a single environment. Applications and services interact by making calls and passing typed parameters — exactly the same way modules interact within an application. The syntax and semantics of interaction is uniform, whether the interacting components are local or remote, and whether they belong to the same user or to different users.

The distinctive features of Hermes are: (1) processes as the basic units of modularity and interaction, (2) ports as "capabilities", (3) a representation-independent, "pointerless" type system, and (4) compile-time checking which enforces protection on the granularity of a module.

We discuss the concept of a multi-application environment. We present the principles underlying the design of Hermes. We summarize the Hermes language, and contrast our design decisions with those used in other languages and systems. Finally, we discuss related research inspired by Hermes.

1 Introduction

1.1 Multi-Application Environments

The traditional computing environment is a single application environment. The early standalone systems, such as the IBM 650 and 1620, were single-application by necessity: they ran one user’s program at a time.

Although batch systems and timeshared systems introduced multiprogramming and a scheduler, they still provided the illusion of the standalone environment by executing multiple standalone environments serially or interleaved. Similarly, most personal workstations provide a simulated standalone environment in each window. Shared services appear as enhancements to one’s standalone virtual machine.

This picture is changing as users perceive the need for distributed applications and for interaction between applications and services outside of the fixed system kernel. In systems such as Mach[ABB+87], applications and services are peer processes that communicate over ports. The system kernel no longer includes shared services, but only the scheduling, communication and protection necessary to support Mach semantics.

While operating systems are evolving towards support for interaction between applications, most programming languages still reflect the assumptions of the standalone or "virtual" standalone environments for which they were designed.

For example, suppose you are developing an application in a high-level language such as C, Modula, or Ada. Your application is running in a virtual address space created by the operating system. You can call procedures in your own address space but not in any others.

To communicate outside your address space, you must interact with an operating system such as UNIX, VMS, or Mach. Your language doesn’t understand the operating system concepts (address spaces, buffers, sockets, pipes, files, etc.), and the operating system doesn’t understand your high-level language concepts (arrays, records, etc.) You must explicitly map the data structures of your high-level language program into low-level objects understood by the operating system. Depending on the sophistication of the operating system, you
may have to cope with memory management, data conversion, naming conventions, recovery from communication failures and processor failures, etc. A simple program in which A calls B and passes it a list can, if B is remote, turn into a much more esoteric program, mostly likely with subtly different semantics. The consequences to the programmer are complexity, lower reliability, and loss of portability.

There are other ways in which high-level languages reflect the assumption that components of a "program" all belong to a single application:

- **Dynamics:** Many languages support only static program configurations. That is, code cannot be added and bindings between modules cannot be changed at run-time. This restriction is reasonable for many (but not all) single-user applications, but is inappropriate for most multi-user applications.

- **Error Recovery:** In many languages (e.g. Fortran, Pascal), an error such as an overflow or an out of range subscript is detected, the program simply halts and terminates. Other languages (e.g. Ada) allow the programmer to provide exception handlers for some of these conditions. However, certain classes of errors, such as undefined variables, are considered too expensive to check. When such errors occur, the result is an erroneous execution with arbitrary effects, which may include program termination. In a single-user environment, termination of your program is annoying but not a disaster — you correct your error, restart your application in a fresh environment, and continue. Obviously, it is unacceptable for errors to have such a global effect if the "program" is an entire collection of applications.

- **Globals:** Many language environments support services which are global to the application. For example, in C, the function `printf` delivers a character string to the "standard output". The assumption that all modules of a program share a single standard output makes sense if by program we mean a single UNIX or Mach task, but not if the program is a multi-application system. However, even languages like Ada, which are intended to support large programs with multiple tasks on machines lacking a general-purpose operating system, include similar global functions.

### 1.2 Hermes Design Principles

Hermes applies the Mach philosophy of multi-application systems within the context of a high-level language.

Our fundamental principles are:

1. **Modularity:** Systems should be built from small independent modules whose interfaces can be written down and enforced.

2. **Uniformity:** Modules interact the same way whether they are local or remote, and whether or not they belong to the same application.

3. **Abstraction:** The high-level language should hide as many low-level details of the underlying implementation as possible.

4. **Analyzability:** Programs should be easy to read and for compilers to check and optimize.

Let us discuss these principles more concretely, to distinguish Hermes from other efforts with different but similar-appearing principles.

Our modularity principle implies that we must enforce protection (what Hoare calls security[Hoare]) at the granularity of an individual module. Other languages assume that the operating system has created a number of separate "address spaces" with hardware-enforced firewalls. In these languages, it is considered acceptable for erroneous modules to cause unpredictable side effects to other modules in the same address space. Even in languages such as Modula 3, where safety and compile-time checking is considered important, it is assumed that parts of programs will need to use unsafe subsets either for efficiency or for expressive power. In Hermes, we aspire to efficiency and expressiveness without the need for unsafe subsets.

Our uniformity principle contrasts to approaches such as Argus[Li82] and Eden[ABL85]. In these approaches, there are multiple interaction modes: messages, and shared atomic objects. Hermes uses messages exclusively, offering one-way messages (send) or two-way messages (call). We avoid shared memory because we want a single interaction mechanism and because we believe that the possibility for interference in shared memory systems inhibits modularity and analyzability. Another consequence of uniformity in Hermes is that there is one kind of module (process), as
opposed to Argus's guardians and handlers, or Ada's
tasks, packages and procedures.

Our abstraction principle looks at first glance like a tru-
ism. However, this principle results in significant differ-
ences between Hermes programs and programs in other
languages. Specifically, Hermes hides the operating sys-

tem, and Hermes gives up "performance transparency".
In C, for instance, there is no facility for multiprocess-
ing, dynamic loading, or even dynamic (heap) memory,
so one must call system services, such as `fork`, `exec`,
`malloc`, to perform these functions. A C programmer
must also be conscious of data layout. In Hermes, the
operating system is hidden underneath the language.
Applications never call the operating system \(^1\). Performance
transparency is the principle that one can esti-
mate the cost of the underlying implementation by
looking at the source code. While performance trans-
parency can be useful under certain circumstances, we
give it up in Hermes so that details of implementations
(such as whether calls are local or remote, or whether
parameters are passed by copying or by reference) can
remain hidden from users, and so that implementers
are free to design more exotic implementations.

The principle of analyzability goes hand in hand with
our principle of abstraction. The Hermes programmer
gives up the ability to write an exotic program with
multithreading, aliased variables, and other properties
which make programs hard to understand. In exchange,
the compiler gains the ability to detect the possibility
of applying an optimization which may introduce alias-
ning and multithreading under the covers. Our con-
jecture is that by providing a simpler high-level model in
which programmers may not write "tricky" code, and a
library of optimizations which reintroduce the "tricks"
into the implementation, we will improve reliability and
at least break even in efficiency.

2 Hermes Language

Hermes is fully described in a tutorial and reference
manual[SBG+89, SBG+90]. The implementation of
Hermes is described in [BL90]. The salient features
are described below.

\(^1\)The only exception to the above is the external input and
output from a Hermes system. This input and output normally
goes to physical devices, but if Hermes is embedded within an op-

erating system, the input and output may be directed at services
within that system, e.g. a window manager or print server.

2.1 Processes and Communication

Hermes is based upon the process model, in which a
module equals a process. The process model incorpo-
rates the useful features of both the traditional Algol
model and the object-oriented model, while consider-
ably simplifying concurrent and distributed program-
ing.

Modules are units of separate compilation and private
data ownership. Processes are threads of control. In
Hermes the two are the same. A Hermes process is an
active entity which includes a set of variables, and a
sequential program. The sequential program executes
by operating on the variables of that process. There
are no shared or global variables. There is no aliasing
— therefore, there is no need to make the distinction
between variables and locations.

Processes interact through ports. Input ports are mes-

sage queues. Output ports are connections to input
ports. (The connections are also called bindings or ca-
pabilities — we will use these terms interchangeably.)

The sole interprocess communication mechanism in
Hermes is message passing over port-to-port connec-
tions. There are primitives for one-way message passing
(sending), and two-way message passing (calling).

The following operations (written using abstract syn-
tax) perform communication:

- `send(m, op)`
- `receive(m, ip)`
- `call(op, [p1, ..., pn])`
- `return(m, [exc])`
- `select([ip1, g1, s1], ..., [ipn, gn, sn])`

The `send` operation removes the value from a variable
m, and sends it to the input port to which the output
port op is currently bound. This operation does not wait
for a reply.

The `receive` operation removes a message from the
queue in input port variable ip, and stores the mes-
sage into variable m. The process will block until a
message is available.

The `call` operation moves the values of variables p1
through pn into a `callmessage`, which it then sends to
the input port to which output port op is currently bound. The operation then waits until the callmessage is returned, after which the (possibly modified) values are moved back into \( p_1 \) through \( p_n \). The returned callmessage may optionally indicate that an exception is to be raised in the caller.

The return operation returns the callmessage currently stored in variable \( m \) to the original calling process. An optional exception \( exc \) may also be returned.

The select operation performs a guarded selection similar to those in CSP and Ada. Variables \( g_1 \) through \( g_n \) are boolean guards; variables \( ip_1 \) through \( ip_n \) are input ports, and \( s_1 \) through \( s_n \) are statement lists. A (possibly nondeterministic and unfair) choice is made to execute some \( s_i \) such that guard \( g_i \) is true and such that a message exists on port \( ip_i \). If all \( g_i \) are false, an exception is raised immediately. Otherwise the process will block until there is a message on some \( ip_i \) for which \( g_i \) is true.

Messages are delivered reliably and fairly. That is, any message sent will eventually be received if the process owning the input port issues enough receive operations. No message will be infinitely often overtaken by other messages.

Processes are scheduled fairly. That is, any process which is live in isolation will make progress when executed together with other processes.

The above semantics is consistent with all our design principles. Modularity is supported because each process has its own local variables and no other process can interfere with these variables. The implementor of an interface only needs to know about the interfaces — the conventions for sending to, receiving from, and calling other processes. Fair scheduling and fair message delivery prevent deadlocks caused by other processes which do not explicitly interact with the given process.

The model is uniform. The same syntax and semantics are used whether two processes are local or remote, and whether the processes are part of the same application or not. Furthermore, the single concept of process subsumes as special cases procedures and data abstractions. A procedure is simply a process which is created for a single call and dies after returning. A data abstraction is a process with one input port \( ip \) per abstract operation. Once initialized, the process enters a loop in which it repeatedly performs a select operation on its input ports, waiting until another process invokes one of the abstract operations by making a call. When the call arrives, the data abstraction process executes a receive, then the code of the "method", and finally a return, after which it repeats the loop.

Our model is abstract. Low-level details do not show through to the user, and there may be a wide variety of implementations. For example, the implementation of a call may depend on whether the called process is local or remote. If the called process is local, the data can be "moved" into the callmessage by reference. Additionally, if the called process is both local and currently waiting to receive a message, the "queueing" of the message, the blocking of the caller, and the "scheduling" of the called process can be implemented by jumping directly to the code of the called process. This optimization yields an implementation of a message send plus process switch which can be as fast as 9 instructions on an IBM 370[BS89]. On the other hand, if the process is remote, the implementation may need to perform data conversion (if the target machine has a different architecture), and must implement a reliable fair message communication.

There is considerable room for other optimizations: for instance, data values may be passed eagerly (simultaneously with the call), or lazily (on demand when referenced by the called process); the data can be moved to the called process, or the process can be migrated to the site of the calling process. Even more exotic optimizations are under study in the Hermes research group — these include: (1) "optimistically" continuing to execute the calling process even though the called process has not returned (guessing that no exception will be returned and that no needed values will be modified, and rolling back if the guess is wrong)[BS90]; (2) transparently replicating the called process on multiple sites to reduce communication delay[GJ87].

2.2 Capabilities

Multi-application environments are dynamic. Applications may be created and destroyed, connections between applications may be created and destroyed, all while other applications in the system continue to run.

Some programming languages support no dynamic binding or only limited dynamic binding. For example, in Fortran all binding is static. In Ada, an entry name is statically bound to a piece of code, but you may dynamically select the process instance in which that code runs. In C, you can use procedure variables to bind to a procedure whose identity is not known at
compile-time. None of the above languages include a mechanism for dynamically loading new code.

Operating systems support dynamic creation of applications (tasks). Each operating system has its own system for naming services, binding applications to services, and permitting control over which applications can access which services.

A particularly flexible technique for binding programs to services is based on the use of capabilities. A capability is a data value representing the right to communicate with a service. Capabilities are unforgeable — therefore you have a capability only if you own the service yourself or if someone else with the capability explicitly gave it to you. Capability-based systems are very flexible, since they let the programmer explicitly implement arbitrary access control policies by writing applications whose function is to hand out capabilities. Mach is a capability-based system in which the port send rights serve as capabilities. Capability-based systems are attractive alternatives to systems in which access control is defined in the kernel or in an installation-wide security module which can only be modified by system administrators and "superusers".

A problem with the implementation of capability-based systems is that if users can store capabilities in the clear in their private address space or file, then they can potentially forge capabilities by overwriting the capability with random junk. In the absence of special hardware to tag capabilities, operating systems must store capabilities in shadow areas accessible only to the kernel, and the kernel must mediate every attempt to invoke a capability.

Hermes is a capability-based system integrated into a programming language. A Hermes output port behaves as a capability, because Hermes has chosen to treat ports as first-class values. Output ports can be stored in variables and passed in messages just like any other values. Furthermore, Hermes ports have two properties which Mach ports lack: (1) they are typed, and (2) they are unforgeable despite being stored in ordinary process memory.

The type of an input port determines the type of message which can be received from that port. If the message is a callmessage, the type information includes additional information about the interface, such as which parameters are constant, which are output only, and which exceptions may be raised. The type of an output port determines the type of input port to which the output port can be connected. This guarantees that the owner of an output port and the owner of the input port agree ahead of time on the interface. Only this information needs to be known statically. Any combination of sender and receiver or calling process and called process may be connected provided that the types match. The choice of which process instance running which process code to bind to a particular output port can be deferred until execution time. In fact, the called process can even choose to pass its input port to a successor process during execution, thereby implementing dynamic "plug-replacement" during execution.

Ports are unforgeable because the compiler ensures that the value of a port variable is a port value of the correct type. It is illegal to store a value of integer type into a port variable, or to use a port variable which has not been initialized and which therefore may have some arbitrary value.

There are only three special operations on ports:

- \( ip \leftarrow new() \)
- \( op \leftarrow connect(ip) \)
- \( op \leftarrow create(pgm) \)

The new operation creates an input port with an empty queue.

The connect operation creates an output port which is connected to the specific input port. A process gives other processes access to an input port it owns by issuing connect and then passing the output port to another process using send, call, or return.

The create operation creates a new instance of a process. The variable pgm is of builtin type Program; it holds the code of the program in abstract syntax form. The result of create is an output port connected to a designated initialization port in the created process. This is one of the few operations in which a run-time type check is needed; it must be the case that the static type of op matches the type declared for the initialization port in the run-time value of pgm.

The five communications operations, and the three dynamic configuration operations are extremely flexible and expressive. With them, programmers can build programs at run-time, store them in repositories, instantiate and execute distributed applications consisting of dynamically loaded components, and write arbitrary access control policies. All this is done without the programmer’s being aware of the operating system.
2.3 Type System

The Hermes approach to programming is to write a correct program first in a simple language abstracted away from the machine. Once the program is correct, it is then tuned by selecting different compiler options or pragmas. These pragmas may suggest alternative data representations, such as storing a set as a doubly linked list rather than an array. Or the pragmas may suggest alternative optimizations, such as replicating process X, or applying concurrency control to process Y. Since tuning only replaces one optimization with another, a bad pragma only results in a less efficient program, not an incorrect program. You don’t risk breaking the program when you tune it, since you are not changing any of the source code. It is also much easier to port such programs to other machines.

This philosophy is reflected in the Hermes type system. The types are very high-level and leave considerable freedom for alternative implementation choices. The most noticeable difference in Hermes’ type system from the perspective of a C programmer is the lack of pointer types. Pointers are useful constructs for implementing many different data structures, but they also introduce aliasing and increase the complexity of program analysis. Instead, Hermes has higher-level types, such as tables, which the compiler then maps down to structures containing pointers.

The Hermes type system is built upon eleven type constructors, which we categorize as scalar types, aggregate types, and communication types.

- **Scalars**: integers, booleans, enumerations, reals, nominals
- **Aggregates**: records, variants, tables
- **Interaction**: input ports, output ports, callmessages

Scalars are similar to the scalars of conventional languages.

Records are similar to conventional records, except there are no assumptions about physical data layout. Variants are safe unions. Tables are aggregates of values of the same type. Tables were designed to be similar to relations in relational databases. In fact, a table of records behaves exactly like a relation. They may optionally be ordered, like strings, or unordered, like sets. There are operations to insert and remove elements, to merge and partition tables, or to select an element or sub-table whose content satisfies a particular predicate.

Interaction types define the interfaces between processes. Each interface determines the type of data exchanged, the possible exceptions which may be returned, and other static information called typesate, which we discuss in the following section.

It is important to note that the type of a port says nothing about the code of the process which will receive the calls sent on that port. This is very different from the decision made in Ada and other “data abstraction” languages. In these languages, there is a one-to-one correspondence between the type and the code body. For example, in Ada, any two objects of the same task type have not only the same entries but the same code. In Hermes, the caller doesn’t know the code body of the called process, only the interface. The decision to “plug in” a particular process is made dynamically.

2.4 Checking

In Hermes we do as much static checking as possible. The purpose is to detect nonsense at compile-time wherever possible, rather than at execution-time.

Accordingly, we require that interface types be explicitly declared, and that all variables except temporary variables be explicitly declared. Hermes checks at compile-time that every operation acts only on operands of the correct type.

Hermes incorporates an additional level of checking — typesate checking — which was used in Hermes’ predecessor language, NIL[SY83]. A typesate is a static property which is invariant at any particular program point, but which varies from point to point. Examples are whether a variable has been initialized, or whether a constraint (e.g. positive integer) has been tested. Every primitive operation has typesate preconditions and postconditions. Every user-defined call interface is declared with typesate preconditions and postconditions.

The Hermes compiler tracks typesates using a dataflow analysis algorithm[SY86]. It also rejects programs whose analyzed typesates do not satisfy the all the preconditions. Here are some program errors detected by typesate checking:

- Uninitialized variable errors
Variant case errors
Failure to check a constraint
Instantiating an unchecked program

Besides catching errors early, typestate checking makes it possible to enforce module level security without extensive run-time checking. Security is obviously important if a single language environment is going to run multiple users. There are also performance benefits, since it is no longer necessary to put different processes in separate address spaces. Communication and context switching can be extremely efficient — as efficient as subroutine calls in conventional languages.

3 Hermes Research

Hermes evolved from an earlier experimental language, NIL[SY83, SY85a]. NIL incorporates typestate, the process model, and relational tables. An implementation of NIL for VM/370 was developed, with the goal maximizing execution-time efficiency. NIL was used to prototype two IBM communications subsystems — the SNA transport layer and the SNA LU6.2 session protocol.

We implemented a NIL compiler and realistic prototypes to address the concerns within the development community that efficient “systems” programming might be incompatible with very high-level languages, compile-time type and typestate checking, and with a programming paradigm based upon one process per module.

We evaluated the NIL prototype by measuring the time to develop, test, debug, and extend the system, the execution speed of the generated code, and the number of programming errors caught by compile-time and run-time checking.

Our experience with NIL was so encouraging both from the standpoint of software engineering and from the standpoint of performance, that we decided to explore how far we could extend the principle that a high-level language should hide low-level implementation detail.

This decision eventually led to two followon efforts: (1) to hide communication and physical distribution, and (2) to hide recovery.

The first effort led to the design and implementation of Hermes. Hermes currently runs on 4.3 BSD on the IBM RT, Sun 3's and 4's running SunOS, and on the IBM RISC System/6000. A port to Mach is planned.

The second effort led to the design of Optimistic Recovery, a transparent technique which makes it unnecessary for programmers to treat process state as volatile and do their own checkpointing and restart ([SY85b, SBY88, SYS87, SY88]). Using Optimistic Recovery, a programmer can program for an idealized virtual machine in which processes never fail. Optimistic Recovery can be used to make any collection of communicating processes recoverable — e.g. a distributed Mach application. Optimistic Recovery enhances the value of Hermes in two ways: (1) the Hermes implementation does not have to reflect machine failure back to the user, which would have introduced an undesirable difference between local and remote invocations, (2) Hermes programs do not have to write their persistent state to files.

As of 1990, our project at IBM Research is exploring additional enhancements to the language design and implementation techniques:

- Exploiting the Hermes model of processes and communication to other languages, to permit interoperability across languages and machine architectures [YGS+89]
- Refinements to typestate checking, which would allow programmers to track more information about the value of variables, and which would allow additional coercions ([SY89, SY90]).
- Optimistic transformations to perform process replication ([GJ87]), call-streaming ([BS80]), and parallelization transparently ([SY87]). Such optimizations make it unnecessary to code these complex protocols by hand.

References

[ABLN85] G.T. Almes, Andrew P. Black, E.D. Lazowska, and J.D. Noe. The Eden system:


