INTERCONNECTING HETEROGENEOUS COMPUTER SYSTEMS

A software structure created by the Heterogeneous Computer Systems (HCS) Project at the University of Washington was designed to address the problems of heterogeneity that typically arise in research computing environments.

DAVID NOTKIN, ANDREW P. BLACK, EDWARD D. LAZOWSKA, HENRY M. LEVY, JAN SANISLO, and JOHN ZAHORJAN

Hardware and software heterogeneity arises in many computing environments for many reasons. In our own environment—an academic department with a significant experimental research component—heterogeneity arises because experimental computer research is often best conducted on a high-level test bed (e.g., Lisp and Smalltalk machines, multiprocessor workstations), and because such research often produces unique hardware/software architectures (e.g., prototype distributed systems, special-purpose image analysis hardware). Our environment currently includes more than 15 significantly different hardware/software systems.

Today's very loose interconnection of heterogeneous computer systems (HCS—see Table I for a list of acronyms used in this article) poses several significant problems. One problem is *inconvenience.* An individual must either use multiple systems or else accept the consequences of isolation from parts of the environment. Isolation is generally unacceptable, so many users regularly work with several systems, through relatively crude techniques such as multiple terminals/workstations or Telnet/FTP. A second problem is *expense.* The hardware and software of the environment are not effectively amortized, making it unnecessarily costly to conduct a specific project on the system to which it is best suited. One must acquire not only the system directly required to support the project, but also the peripheral hardware and software necessary to allow that system to function as a largely independent entity. A third problem is *diminished effectiveness.* On many projects, substantial effort must be diverted to address the problems of heterogeneity; time-consuming hacks by scientists and engineers who should be doing other work are the rule, rather than the exception.

TABLE I. Glossary

BIND	Berkeley Internet Domain Server
HCS	Heterogeneous computer systems
HMS	HCS mail service
HNS	HCS name service
HRPC	HCS remote procedure call
IDL	Interface description language
MSM	Mail semantic manager
NSM	Name semantic manager
RPC	Remote procedure call
THERE	The HCS environment for remote execution
TPL	THERE programming language

The widespread availability of communication protocols such as TCP does not solve these problems, because constructing new services and applications on top of such protocols is too difficult. File transfer and remote terminal programs also are insufficient, since their users must explicitly manage multiple machines.

Our particular approach to accommodating heterogeneity [10] is motivated by several widespread (though certainly not universal) characteristics of our environment. We have a large number of system types, but only a small number of instances of some of these system types. New system types are added relatively often. System types are acquired precisely because of their

This material is based on work supported by the National Science Foundation under Grants DCR-8352098, DCR-8420945, and CCR-8611390, by an IBM Faculty Development Award, by the Xerox Corporation University Grants Program, and by the Digital Equipment Corporation External Research Program.

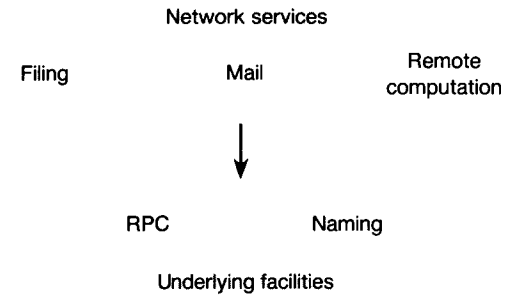

FIGURE 1. Relationship of HCS Facilities and Services

ability to support specific projects, without regard for whether heterogeneity is increased. Thus, it must be possible to incorporate a new system type into the environment at low cost and without masking the unique properties for which that system was obtained. This leads to an approach that we refer to as "loose integration through shared network services." As illustrated in Figure 1, two underlying facilities—remote procedure call (RPC) and naming—support a set of key network services that are adapted to the demands of a heterogeneous environment: remote computation, mail, and filing. Our approach can be characterized in a bit more detail as follows:

- We provide a set of network services that are made available to a heterogeneous collection of client systems through the use of our RPC and naming facilities. The services we have selected are those fundamental to cooperation and sharing.
- In both the services and underlying facilities, we attempt not to legislate standards, but rather to accommodate multiple standards. This allows the integration of unmodified or minimally modified systems.
- We do not attempt to provide existing programs with transparent access to services. The primary reason is that we have too many system types to permit cost-effective development and maintenance of the modifications necessary for transparency. A secondary reason is that transparency is impossible for systems in which the source code is unavailable. Transparency can be provided for specific system types if sufficient use exists to warrant the investment and if the required source code is available.
- We focus on system heterogeneity rather than language heterogeneity. Just as we cannot generally justify the effort to support transparent access to services, we cannot commit to providing complete integration of programs written in different languages. Existing solutions to the "easy 80 percent" of the language heterogeneity problems will suffice for our purposes.

Each of the individual facilities and services reflects this approach.

- The HCS RPC (HRPC) facility utilizes a modular design that, by appropriate selection of implementations at run time, can be made to emulate a wide variety of existing RPC facilities. Thus, the central core of HCS—those systems on which the HRPC facility has been implemented—can easily be adapted to communicate with a new system type.
- The HCS name service (HNS) creates a global name space that accesses names and data from existing name services. By using data in existing name services, rather than reregistering data into an entirely new name service, existing clients can work with their name services without change, and new clients of HNS need not make changes when a new underlying name service is introduced.
- The HCS remote computation service provides a generic mechanism by which services can be executed remotely. Each remote service includes a description of its required inputs and outputs, the steps needed to process the information, and the steps required to create an environment in which to execute the service. These descriptions are processed by interpreters that are responsible for passing information between nodes and for performing any necessary translation of file names, options, etc.
- The HCS mail service (HMS) attempts to improve the quality of most existing mail services while integrating services that are based on diverse models. The mail service is structured like the Xerox Grapevine mail service [9], but also integrates mail systems such as UNIX's® `sendmail` [1]. Abstract mail retrieval and submission interfaces are defined and implemented in multiple ways, facilitating the integration of new mail systems.
- The HCS filing service is represented by two distinct efforts: The first approach defines a centralized filing service that stores files in multiple representations (based on those used in the HRPC facility). The second approach is based on that of the naming facility, where existing local file systems are used to store data, and neither the files themselves nor information about them (such as the file type) need be reregistered.

The role of our work has been to devise approaches, produce designs embodying those approaches, and test these designs through prototype implementations. This article does not represent a "quick packaged universal solution" to every person's heterogeneity problems. On the other hand, the ideas presented here represent more than "academic speculation," since implementations exist in each area, some of which have stood the test of production use by nonsympathetic audiences.

In related work, other styles of heterogeneity demand somewhat different solutions. MIT's Project Athena [4] and Carnegie-Mellon's Information Technology Center (ITC) project [31] are two highly visible efforts. Each seeks to accommodate heterogeneity

UNIX is a registered trademark of AT&T Bell Laboratories.

through *coherence*: enforcing high-level uniformity in software while permitting implementation on diverse hardware. Both projects rely primarily on UNIX. Project Athena is standardizing on an applications interface, and ITC on a centralized file service.

Another major effort is the MIT Laboratory of Computer Science (LCS) Mercury project, which attempts to share programs written in substantially different languages such as Lisp and CLU. The LCS group hopes to provide a "semantic bridge" between these languages.

The UCLA Distributed Systems Laboratory is concerned with integrating computational resources with a high degree of transparency. In one approach, they developed LOCUS [45], a single distributed operating system that runs on multiple, heterogeneous machines, including VAXes, IBM 4300s, and IBM PC-ATs. In an alternative approach, they are developing transparent operating systems bridges [20] with the goal of integrating machines with dissimilar operating systems.

General Motors' MAP (manufacturing automation protocol) [3, 27] is an industrial effort to achieve coherence through standardization. MAP is based on the seven-level ISO standard and specifies protocols from the physical interconnect level to the application level. At the physical level, a MAP network is based on a 10 Mbit/s, broadband, token-based, coaxial cable. Device-to-device communications may require conformance to lower layers only, while application-to-application communication will typically require all seven levels. MAP's goal is to encourage vendors to supply hardware and software using MAP protocols. Thus, MAP attempts to solve the problem of heterogeneity by enforcing homogeneity of communication.

REMOTE PROCEDURE CALL

Network communication is the sine qua non of our work. Although some form of networking capability is possessed by all systems of interest to us, no single protocol is shared by all of them. Even among each subset of systems that share a protocol, the precise function of and interface to network operations can differ substantially. This absence of protocol standardization is one serious impediment to accommodating heterogeneity.

A second, equally serious problem is that, until recently, commercially available network implementations provided only low-level services. Higher level functions are generally encapsulated in application programs such as Telnet, FTP, and NSChat. The absence of low-level protocol standardization, however, makes it particularly important that application code be insulated from this layer. Furthermore, building applications on top of low-level services is beyond the capabilities of most programmers.

One attractive approach to this problem is RPC [8]. An RPC facility provides a user-level mechanism across the communication network that, as much as possible, has the same syntax and semantics as local procedure calls within the application program's high-level language. Hence, RPC supports communication among application programs while relieving programmers from concern with data encoding, transport protocol details, etc. The run time system of an RPC facility is responsible for mapping the language's calls and high-level type system into the facilities provided by the low-level network protocols. Although most RPC implementations exhibit limitations when measured against the demands of a heterogeneous environment, the RPC model itself has various characteristics that make it an ideal vehicle. One of the few areas of consensus at the 1985 "ACM SIGOPS Workshop on Accommodating Heterogeneity" was the appropriateness of RPC in a heterogeneous environment [32].

To a first approximation, an RPC facility works in the following way: The client (caller) and server (callee) modules are programmed as if they were intended to be linked together. A description of the server interface, that is, the names of the procedures and the types of arguments the server implements, is processed, yielding two *stubs*. The client stub is linked with the client; to the client this stub looks like the server. The server stub is linked with the server; to the server this stub looks like the client. The stubs shield the client and server from the details of communication.

The construction and use of an RPC-based distributed application can be divided into three phases: *compile time*, *bind time*, and *call time*. Compile time involves the production of stubs, which ideally is done mechanically by a stub generator that processes an explicit definition of the interface, written in an interface description language (IDL). Bind time involves the server making its availability known by exporting itself, and the client associating itself with a specific server by making an import call to this mechanism. Call time involves the *transport protocol*, *control protocol*, and *data representation*. The transport protocol is used by the stubs to convey arguments and results reliably between client and server. The control protocol consists of information included by the RPC facility in each transport packet to track the state of the call, which may require multiple transport messages. The data representation is a convention for ensuring data compatibility between client and server (e.g., byte ordering or record layout).

Existing RPC facilities make significantly different choices in each of these five areas: compile time support (including the programming language, the IDL, and the stub generator), the bind time protocol, and the three call time protocols—transport, control, and data representation. Although in principle these choices are orthogonal to one another, in practice they are intertwined in each implementation. As a result, the various existing RPC facilities not only are incapable of communicating with one another, but are also difficult to modify to make such communication possible.

This need not be the case. As one example, an RPC facility implemented at the DEC Systems Research Center is able to employ different transport protocols between different pairs of systems (personal communi-

cation by A. D. Birrell, 1984). This was accomplished by two steps: First, a clean interface, consisting of three procedures, was defined between the stubs and the transport mechanism; these procedures could be implemented in several different ways by placing a thin veneer over common transport mechanisms. Second, binding was augmented to include a mechanism for determining which transport protocol should be used between a specific client and server, and to return the correct three procedure implementations to the stubs. Thus, the same mechanically generated stub could employ a variety of transport protocols, with the choice delayed until bind time.

Inspired by this modularization, in our HRPC facility [7] we have specified clean interfaces among all RPC components. An HRPC client or server and its associated stub can view each of the remaining components as "black boxes" that can be mixed and matched. The set of protocols actually used is determined dynamically at bind time—long after the client or server has been written, the stub has been generated, and the two have been linked. This design meets two key objectives: We are able to *emulate* existing RPC facilities by providing appropriate implementations of the underlying abstractions, thus allowing *unmodified* native RPC systems to communicate with our core HRPC systems; and we are able to employ existing software (e.g., transport protocols) easily in building an RPC facility for a new system that does not have a native facility.

Two examples illustrate our approach. The first is the Face–Finger Service (`ffinger`), a relative of the Berkeley UNIX `finger` program. The `finger` program returns textual information (full name, phone number, etc.) about a given user on a given machine. The `ffinger` program provides a heterogeneous, distributed, department-wide service that provides pictures of each user. When a user `ffingers` another user, a window pops up on the screen with the picture, full name, office phone, and so on, in a "bubble-gum baseball card" format (see Figure 2).

The server for `ffinger` runs on a single machine and is implemented using HRPC. Clients who call the server and display the baseball cards have been implemented for workstations including VAXes, SUNs, Xerox D-machines, Tektronix 4404/4405s, and IBM RT/PCs. Each of these clients uses its own native RPC facility and windowing system; each believes it is communicating with a server written using the same native RPC facility. HRPC creates this illusion and allows the server to deal simultaneously with a variety of clients.

The second example illustrates the case of a single client using multiple servers. We designed a server that returns a list of users logged in to the machine on which it resides. We implemented this server on three different systems: on Xerox computers using the standard Xerox RPC (i.e., the XNS protocol for transport and the Courier protocols for binding, data representation, and control) [46], on SUN computers using the standard SUN RPC with UDP data grams (i.e., the UDP protocol for transport, the XDR data representation standard, and the SUN protocols for binding and control) [41, 42], and on VAX computers using the standard SUN RPC with TCP (i.e., the TCP protocol for transport, the XDR data representation standard, and the SUN protocols for binding and control). We then implemented an HRPC client of this service. This single client can bind to each server using that server's own native binding protocol and communicate with each server using that server's own native RPC; the same client can make a sequence of calls to different servers, each call emulating a different native RPC.

The Call Time Organization of HRPC

In traditional RPC facilities, all decisions regarding implementation of the various components are made when the RPC facility is designed. Making these choices early simplifies the work done at run time. For instance, in such systems the only information needed by a client to access a server is the location of that server; no other decisions concerning details of communication between client and server need be made.

Acquiring this location information is the process of *binding*. Executing the HRPC binding protocol yields a `Binding`, a data structure containing information de-

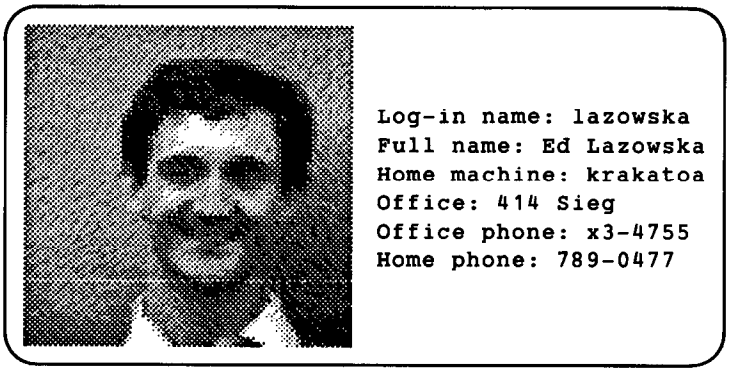

FIGURE 2. Display from the Face-Finger Program

scribing the logical connection to a server. A `Binding` typically is held by the client and passed to the stub as an explicit parameter of each call.

The basis of the HRPC factorization is an abstract model of how *any* RPC facility works, expressed through a procedural abstraction of the call time components (transport, control, and data representation). Each HRPC stub is written in terms of these abstract interfaces. During binding these interfaces are bound to implementations, selecting a specific combination of control protocol, data representation, and transport protocol components. In addition to the location information, a `Binding` explicitly represents the choices for these three components as separate sets of procedure pointers. At call time, references to the component routines are made indirectly via these procedure pointers. The interaction between these entities is depicted in Figure 3, where the direction of the arrows indicates the direction of calls during the call portion of an RPC. Returns are made in a reverse manner, with messages containing results passed back from the server to the client.

The Bind Time Organization of HRPC

The first step in binding is *naming*: the process of translating the client-specified server name into the network address of the host on which the server resides. The second step is *activation*: Some RPC designs assume the server is already active; others require that a server process be created dynamically. The third step is *port determination*: The network address produced during naming does not generally suffice for addressing the server, since multiple servers may be running on a single host. So, each server is typically allocated its own communications port, which, together with the network address, uniquely identifies the server.

The naming and port information constitute the location of the service. The client's outgoing messages can use this location information, and the server can reply using information passed up to it from the transport level upon receipt of an incoming call.

Consider the case of an HRPC client importing a server written using some existing RPC. The client specifies a two-part string name containing the type (e.g., `FileService`) and instance (e.g., a host name) of the service it wishes to import. To honor this request, the HRPC binding subsystem first queries the name service (described under "Naming"), retrieving a `Binding Descriptor`. Each `Binding Descriptor` contains a machine-independent description of the information needed to construct the machine-specific and address-space-specific `Binding`. In particular, a `Binding Descriptor` consists of a designator indicating which control component, data representation component, and transport component the service uses, a network address, a program number, a port number, and a flag indicating whether the binding protocol for this particular server involves indirection through a binding agent. The remainder of the `Binding` must now be completed in accordance with the information in the

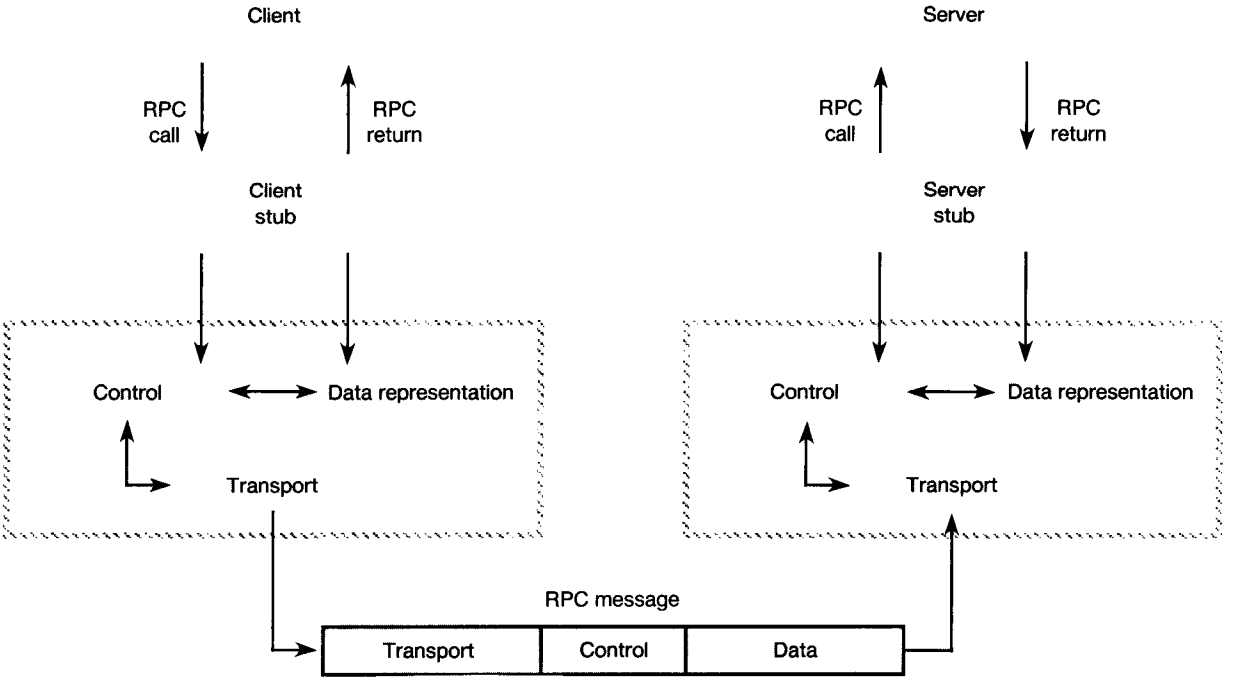

FIGURE 3. Interaction among Call-Time Components in HRPC

`Binding Descriptor`. To do this the procedure pointer parts of `Binding` are set to point to the routines to handle the particular control protocol, data representation, and transport protocol understood by the server.

The HRPC Stub Generator

Stubs insulate the code that actually implements the functions of the RPC client and server from the details and complexity of the RPC run time system. To facilitate this the IDL specification of an RPC service is processed by a stub generator. The generator consumes the specification, which consists of procedure names and the data types of their arguments and results, and produces appropriate stubs in a designated programming language. These stubs are compiled and linked with the actual client and server code.

The HRPC system uses a stub generator for an extended version of the Courier IDL [46], based on the generator written at Cornell [24]. The "code generator" portion of the stub generator was heavily modified to support the HRPC interface. The stub routines are generated in the C programming language. Our major addition to the Courier IDL is an escape mechanism known as `USERPROC`, which allows users to provide their own marshaling routines for complicated data types, such as those containing pointer references.

Although not a primary objective of our work, multilanguage support has been provided by integrating HRPC with the Franz Lisp system [19] running on VAXes and with the Smalltalk-80 system running on Tektronix workstations. Other aspects of HRPC include a lightweight process mechanism and a mechanism by which servers can make calls back to clients.

HRPC's unique hypothesis is that the most effective way to provide basic communication with a diverse set of systems is to emulate the native RPC facilities of these systems. The major intellectual task in HRPC was defining the interfaces between the various RPC components that make this emulation feasible. The resulting modularization has the added benefit of making a subset of HRPC an excellent candidate for porting to a new system that lacks a native RPC, since any existing building blocks can be employed.

A natural concern is that the widespread use of indirection within HRPC might significantly increase execution time. Our benchmarks show, however, that the performance of HRPC is competitive with that of the native RPC facilities being emulated. The reason is that these native RPC facilities use a large number of internal procedure calls for reasons of software structuring: They pay the same price as HRPC without gaining any run time flexibility for it. Of course, these native systems could be streamlined (a few research RPC systems have been), whereas HRPC could not.

HRPC can be viewed as an easy-to-satisfy standard. Given a new system with an existing RPC facility, the addition of new modules to HRPC will make communication possible. Given a new system without an existing RPC facility, the implementation of any one combination of the components emulated by HRPC will make communication possible. HRPC has been in active use for several years and represents a proven approach to the problems it addresses.

In related work, a variety of RPC facilities support differing degrees of heterogeneity. SUN RPC [41] supports the two common byte orderings of integers and two transport protocols. Matchmaker [25, 26] is based on a single operating system that may run on different machines. Messages are tagged to tell the recipient what source-machine representation is used in the message body. Several programming languages are also supported. Horus [18] supports a single RPC mechanism, but for multiple languages. Differences among source languages and machine-specific data representations are embodied in specifications that together with the interface description are inputs to the Horus stub generator. Mixed-language programming (MLP) [22] focuses on the construction of programs with procedures written in different programming languages. MLP is concerned, in part, with accommodating existing programs without requiring that interface specifications be given.

Perhaps most related is the Apollo Network Computer System (NCS) [16], an effort to define a multivendor communications architecture for network services. The basis of NCS is an RPC system similar in several ways to HRPC. NCS/RPC supports both data representation and transport heterogeneity. RPC messages are self-defining, containing information about the types of the data being sent; stubs are independent of any specific format. Stubs are defined in terms of a socket-style transport interface, which can be implemented in several ways; currently, UDP/IP and Apollo Domain transports are supported. NCS/RPC does not handle different control protocols; specifically, NCS/RPC talks only to other implementations of NCS/RPC implemented on a heterogeneous set of systems.

NAMING

Name services [29, 33, 44] provide the run time mapping of names into data. For the most part, a *name* is simply a character string that conveniently allows a human to identify a resource. For instance, `samar.cs.washington.edu` is an ARPA domain-style name [34] for a VAXstation-II host. The *data* associated with a name can be almost anything, but most often involves information that is likely to change infrequently. The most common use of name services is to obtain addresses. For example, the Berkeley Internet Domain Server (BIND) name service [44], which supports domain-style names, contains a mapping from `samar.cs.washington.edu` to the IP address 128.95.1.32. Hosts wishing to communicate with `samar.cs.washington.edu` obtain its address dynamically by querying BIND. This run time determination of addresses simplifies the management of distributed systems, since each host may be administered individually, including changes to its location. All that

is required for continuing operation is that any location change be registered with the name service when it occurs.

In a heterogeneous system, it is necessary to manage a *global name space*, that is, a set of names whose associated data can be accessed from anywhere in the environment. This global name space allows sharing of names among clients on different systems and is crucial in supporting location-independent execution. It is also necessary for convenient use of the system by human users, as it permits the exchange of names across system boundaries.

Our environment places three specific demands on a name service: First, existing applications must continue to run unaltered. Second, new applications written to use the global service must have access to the naming information contained in newly integrated systems, without requiring recompilation or relinking. Third, the incorporation of new systems must have relatively low development cost.

To some extent these goals conflict with one another. Continued execution of existing applications requires that names be accessible in the existing name services local to the individual systems. Graceful integration of new systems into the global name space might most naturally be accomplished by reregistering the data contained in the local name services in a global service, which carries with it the difficult problem of maintaining consistency between local and global copies. Additional problems that arise due to heterogeneity concern *name syntax* and *name conflicts*. Name syntax is a problem because the separate systems that comprise the heterogeneous environment are likely to have conflicting name syntaxes, so it is not possible to impose a single syntax for the global name space that would be "natural" on all systems. Name conflicts arise because several systems may have identical names that are unambiguous when issued in an environment consisting of only one system, but ambiguous when the systems are combined.

The HCS name service (HNS) is the global name service we have constructed to address these problems [37, 38]. Primarily because of problems of consistency, we have chosen not to perform reregistration in HNS, but to use the local name services directly to store the data associated with the global name space.

STRUCTURE OF THE HNS

The HNS provides a global name space accessible in a uniform manner throughout the heterogeneous environment, and a facility to associate data with those names. Rather than directly storing the data associated with a global name, the data are maintained in an existing name service, where they are associated with some name local to that name service. Viewed at the highest level, the HNS provides mappings between the global name for an object and the name of that object in its local system, while the local name service performs the final name-to-data mappings.

Each HNS name contains two parts, a *context* and an *individual name*. The context portion of an HNS name determines the specific name service used to store data associated with that name. The individual name component determines the corresponding local name with which the data are associated in that service. In the simplest case, the individual name is simply equal to the local name, although more sophisticated schemes are allowed. The HNS name for an ArpaNet host might have a context `BIND-hosts` and an individual name `samar.cs.washington.edu`.

Although the HNS does not impose any restrictions on the syntax of individual names, it is required that there be an invertible mapping between individual names within a context and the names of objects in the local name service. This ensures that HNS names are conflict-free. Because the HNS guarantees that only a single name service maintains information on objects in any one context, it is not possible for distinct name services to create name conflicts in the HNS.

Another problem that arises is that local name services may store equivalent data in different formats or store similar but not identical information of a particular type. For instance, BIND and Clearinghouse [33] servers both contain host machine name-to-address mappings, but accessing and decoding this information are done differently in each case.

Ideally the HNS should insulate the client program from this semantic heterogeneity. In particular, it would be unmanageable to require every client program accessing the HNS to understand the semantics of each underlying name service. If this were the case, every client would have to be modified and recompiled whenever a new system type was added to the environment.

The HNS cannot relieve client programs of the burden of understanding the interfaces, formats, and data semantics of the local name services without having some name-service-specific code to support this. In the HNS this code is encapsulated in a set of HRPC-accessible routines called name semantics managers (NSMs). Each type of HNS query is supported by a set of NSMs with identical interfaces, with one NSM for each local name service. (Figure 4 contains a diagram of the logical structure of the HNS.) Because they are remote procedures, NSMs can be added to the system dynamically, without recompilation of any existing code. When a new system is added to the HNS, we construct NSMs for that system and register these NSMs with the HNS; the data stored in them become available through the HNS. Individual data records in the name services are not registered explicitly with the HNS.

Although the number of NSMs grows in proportion to the number of query and system types, it is impossible to avoid this complexity. It is an inherent aspect of a heterogeneous environment that code to deal with each different system type must exist somewhere. We believe that the structure we have created, with NSMs separate from the HNS itself, is the most natural one in terms of ease of use and management.

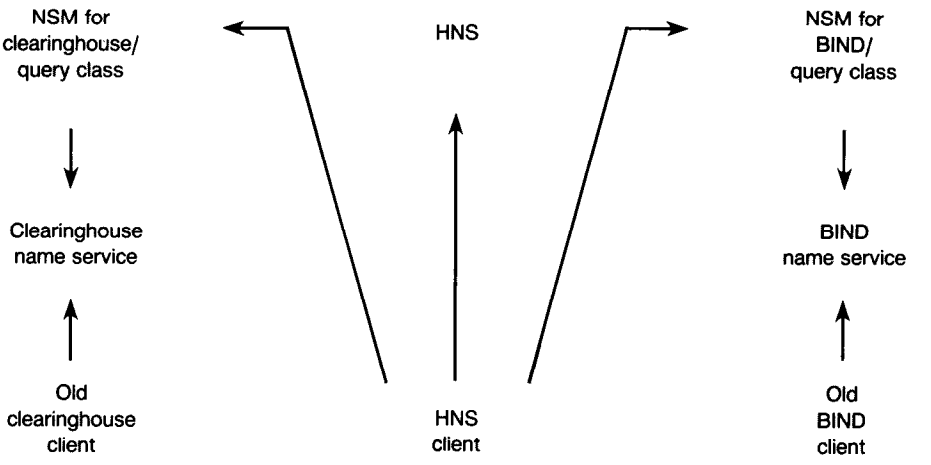

FIGURE 4. HNS Query Processing

HNS Name Lookup Procedure

The logical structure of the HNS and the process followed in satisfying an HNS request has two phases (see Figure 4). In the first phase, the client program calls the HNS using HRPC. The client passes the HNS a name and a query type indicating the type of data desired about that name (e.g., mailbox or telephone number). The HNS uses the query type and the context portion of the name to determine which NSM is in charge of handling this query type for this context. The HNS then returns a `Binding` for the NSM to the client.

In the second phase, the client uses the returned `Binding` to call the NSM, passing it the name and query type parameters as well as any query-type-specific parameters. The NSM then obtains the requested information, usually by interrogating the name service in which it is stored (although any technique, such as looking in a local file, may be employed). This information is then returned to the client. (Figure 4 shows two ways—through BIND or through Clearinghouse—that the second phase can be satisfied, depending on the name passed to the HNS in the first phase.)

Every name provided in an HNS request is fully qualified. Any abbreviations or nicknames must be transformed to fully qualified names by other utilities before they can be passed to the HNS. Especially in large or highly heterogeneous systems, the user may require considerable help in dealing with the global name space, making this separation appropriate.

Separating the NSMs from the HNS incurs the expense of an additional HRPC call over a scheme where NSMs are part of the HNS. Nevertheless, this separation is essential to managing long-term growth, as it separates the query-type-specific interfaces from the HNS and moves them out to the more easily managed NSMs.

Although the above procedure indicates the logical process followed in satisfying HNS requests, an implementation need not be structured in just this manner. For specific query types, it is possible to trade a decrease in management convenience for an increase in performance by making particular NSMs local to either the HNS or the client code. For instance, NSMs that support HRPC binding might be contained in the same address space as the HNS, permitting a client query to be satisfied with a single HRPC call from the client to the HNS and a single local procedure call from the HNS to the appropriate NSM. Alternatively, a particular client might determine that a specific query type is crucial to its performance. In that case copies of the corresponding NSMs could be placed in the client's address space, so that local rather than remote calls to them can be made. Finally, caching can be employed to improve response times to requests [39, 43] at the cost of increased code complexity.

Prototype Implementation

Although the HNS is logically a centralized facility, its implementation must be distributed and replicated for the usual reasons of performance and availability. Because the implementation problems associated with these properties are for the most part successfully addressed in existing name services, we choose to ease our implementation effort by making use of an existing name service in storing the HNS's map from contexts and query types to NSM information. In particular, a modified version of BIND is used for this purpose. The key modification allows dynamic updates, since standard BIND maintains only static information that is loaded from a file when the program is initialized.

The prototype HNS implementation supports a limited number of query types, including HRPC binding. This implementation illustrates the utility of several aspects of the HNS design. First, the use of existing data rather than reregistered data has the advantage that

services running in the underlying systems are generally available to HCS clients without the need to register new information with the HNS. Because of this, client programmers can conveniently make use of other network services, while changes and additions to the underlying service registration information are automatically reflected through the HNS to HCS clients, alleviating problems of consistency and scalability.

In addition to these client-perceived benefits, the HNS provides implementation-level structure for dealing with the heterogeneous naming semantics of HRPC binding in a modular fashion. This structure is manifest in the NSM code for doing binding using each of the underlying name services (BIND and the Clearinghouse in our prototype implementation). Each of these modules deals with a single name service and a single query type, rendering their code manageable and easing the task of adding new RPC and name services to the existing network.

REMOTE COMPUTATION

One advantage of a distributed system is its potential for resource sharing. Availability of a single network resource can remove the need for replicating that resource on each computer. A typical network includes resources of several types, including computational resources such as high-performance processors, input/output resources such as mass storage or printing facilities, and software resources such as special-purpose programs.

Remote computation—the remote execution of a program or service—is one means by which these resources can be accessed. Access to network resources can be provided in several ways. A common scheme is to grant each user an account on each system; tools such as remote login and network file transfer programs allow manual use of facilities. There are three problems with this approach. First, providing accounts on each system may be difficult; it should be possible to permit a user to access a service without providing general access to the machine on which it runs. Second, a user is required to understand the operation, command language, etc., of each of the systems used. Third, users waste a significant amount of time executing remote login and file transfer programs to access remote facilities.

Many systems now provide tools to simplify remote access and aid in remote computation. For example, UNIX system commands such as `rsh`, `rcp`, and `rdist` simplify distributed access within a network of UNIX machines, particularly when used together with shell scripts. Integrated networks, such as Locus [45], Apollo Domain [30], Newcastle Connection [12], VAXclusters [28], and Eden [2], provide transparent access to network computing or storage facilities. Several other systems provide a framework for defining specific network servers and remote clients; for instance, UNIX Maitre'd [5] and Xerox Summoner [21] are both capable of locating lightly loaded servers (for load sharing), transferring files, etc. Dannenberg's Butler system [15] similarly considers these issues, as well as more general problems of resource and access control.

These tools and services all operate within a homogeneous hardware or operating-system environment. In a heterogeneous environment, remote computation becomes both more desirable and more difficult. It is more desirable because, by definition, the network contains systems with different capabilities that users wish to exploit. It is more difficult exactly because of this variety.

Remote computation facilities must solve a number of problems, regardless of the degree of heterogeneity. The creator of the client interface must be able to pass command options to the service. The client must send, or the server must request, files that are needed for execution. Locating files may be complicated, particularly when file names are not explicitly specified in the command string. For example, some text processors maintain auxiliary files that describe the structure of a document; these are read if they exist and are created otherwise. More troublesome, there may be input requirements that become known only during execution.

Some problems in remote computation are more specific to heterogeneity. One fundamental problem is the naming of objects. For example, the structure of file names may differ on the client and server machines. This may include the syntax of file names, the specification of directories, and the specification of devices or even machines on which those files reside. Conventions for naming may be different; compiling the program `myprog` may produce `a.out` on one system and `myprog.obj` on another.

Even describing the service to execute is complicated by heterogeneity. For example, most application programs permit the specification of options. The syntax and semantics of the options will differ on different systems; optimized compiler output may be specified by following the compile command with `-O`, with `/optimize`, or even by selecting a menu item. In addition to the options specified when a program is run, its execution often depends on its environment; that is, contextual information provided by the user and the operating system, including logical names or aliases, a default directory, a directory search path, and some invisible files used by the service for input and output. Each system has a different environment that must be communicated between the client and server.

Another problem typical of any heterogeneous communication is translation of data. Translation must occur on the client side, server side, or both sides, depending on the approach taken. This translation can be handled at a level below the application, as demonstrated by our HRPC system.

Finally, error handling will differ on the client and server. In particular, error messages generated on the server may be nonsense when read in the context of the client. A remote computation system must be aware of the possible error conditions so that they may be reported sensibly to the client.

The HCS Environment for Remote Execution

Our goal is to simplify the construction of remote services in a heterogeneous environment. A service may run on multiple system types or only on a single system. In either case, however, it should be easy to access the service through a collection of heterogeneous clients. We require that, wherever possible, *no* modification of service program code be needed to make that service remotely accessible. The reasons for this goal are that sources for some services (e.g., compilers or formatters) may not be available and that maintaining modified sources for a variety of implementations over multiple releases would present serious problems.

From the user's point of view, the interface to a remote service should be identical to the interface for a local service. Thus, if a service is available both locally and remotely, invocation of the service should use the same command syntax, option names, etc., even if the remote system requires different syntax. One visible difference may be that some services will execute asynchronously with respect to the user's session, perhaps even running in the background.

We have designed and prototyped a remote computation system called THERE: the HCS environment for remote execution [6]. THERE is a facility for constructing remote execution clients and servers in a heterogeneous network. THERE simplifies the addition of new network services and aids the service developer in handling some of the problems mentioned above, including communication of command information, name translation, and file transfer.

The basic structure of THERE is shown in Figure 5. Both client and server execute copies of the THERE interpreter—a generic front-end. On the client side, the interpreter provides the communications path to all available THERE network services. The interpreter parses the user's command line and sends any needed data to the appropriate server. On the server side, the interpreter manages a remote computation session with all services available on a particular node. The server-side interpreter receives requests from clients, establishes the appropriate execution environment, and executes the service or spawns a task to do so. The server determines needed files and requests them from the client through special function calls. File requests and file data are shipped using the HRPC mechanism. Client and server interpreters are nearly identical with the exception of system-local functions and the knowledge of which role is being played. Of course, they may have different implementations on different architectures.

To make a new service available on a THERE server machine, the service builder must first decide what information is needed from clients, what processing will need to be done locally, what environment will be needed for execution, and what data will be returned to the client. Based on these decisions, the service builder codes a THERE programming language (TPL) program, which is a high-level description of the service. The TPL program defines the information to be exchanged between server and client, the steps needed to process that information, and the steps required to create an appropriate environment to execute the desired service. A different server-side TPL program must be created for each system type on which the service runs.

Similarly, a client TPL program exists for each client system that can access a service. When a user issues a remote computation request, the appropriate client TPL program is selected by the interpreter. That TPL program processes the command line, gathers environmental information, defines input/output relationships, and communicates that information to the server.

The information exchanged between client and server is determined by variables that are exported by the client and imported by the server. The server TPL program for a specific service declares a set of variable names, for example, `InputFileName` and `OptimizeSwitch`. The corresponding TPL client program declares similar variable names and binds invocation-specific values to those variables. When the interpreted client TPL program has completed its

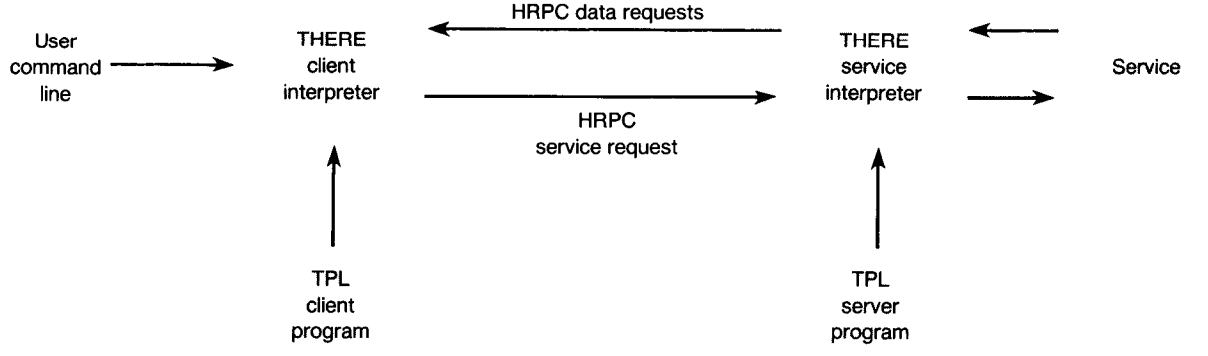

FIGURE 5. Structure of THERE

processing, it tells the interpreter to execute the requested service remotely. The interpreter then uses the HRPC service to send exported variables and their values to the server-side interpreter. One parameter of the HRPC call specifies the requested service so that the server interpreter will know which TPL program to execute.

TPL provides a number of standard programming language features, for example, the ability to loop, compare and branch, build lists, and process strings. The TPL program may also specify local execution of programs to pre- or postprocess files on either the client or server side. Furthermore, TPL contains a number of functions specific to processing remote computation requests in a heterogeneous system. For example, there are built-in functions to create local file names of various file types. Typically, a server will receive file names from the client and must create system-local names with which to store those files. The server must remember the relationship between the client name and the server name, and must also associate created output files with the input files from which they were constructed. In this way the interpreter can produce the reverse mapping from server output file name back to a client output file name.

THERE has been used to construct a number of remote services, including one that successfully supported an undergraduate course by providing remote access to graphics capabilities on IBM PC/RTs, and another that allowed students in graduate courses broad access to an Ada compiler that ran only on a single node.

MAIL

Electronic mail is perhaps the most ubiquitous heterogeneous service. The number of users interconnected by an amalgam of networks is enormous and increasing rapidly. Given that most mail systems are already interconnected, why are we providing a mail service? Despite this general interconnection, the mail service provided to users across systems is often primitive relative to that provided within individual systems. For instance, replying to users is often unsuccessful due to differences in addressing syntax. The HCS mail system (HMS) [40] improves the quality of the mail service among interconnected systems, while easing the integration of new mail systems.

There are two major models of distributed mail services: *host-based* and *server-based*. The host-based model represents the services that have evolved from the original messaging services on time-shared hosts. The most distinctive characteristic of this model is that user messages are maintained by the mail system in files on the local host; network facilities transfer messages between hosts. The services on UNIX and VMS machines are examples of the host-based model. The server-based model represents the services that have been constructed explicitly for networks of computers. The most distinctive characteristic of this model is that user messages are maintained by the mail system on special mail servers. The Grapevine [9] service is a prototypical example of the server-based model.

Attempts to solve many of the problems of heterogeneity in the host-based model are best characterized by the `sendmail` internetwork mail router [1]. The `sendmail` program relies heavily on a configuration file that encapsulates information about heterogeneity. For example, the configuration includes a set of address rewriting rules that define translations to be applied during name resolution, thereby supporting resolution of addresses with different syntax. The evolutionary nature of the host-based model, however, results in a mail service that falls short in three key ways: *addressability* (it is hard to determine the addresses of recipients, since addresses in the host-based model include information about the host to which the mail should be delivered), *availability* (a single host—the one on which a user receives mail—controls the availability of that user's mail), and *accessibility* (users must login remotely to read mail while using a machine that is not their mail host).

The server-based model lessens the problems associated with the host-based model. For instance, in Grapevine each user is associated with a set of mail servers rather than an individual host. Mail is delivered to *any* available server in the set; mail is received by querying *all* available servers in the user's set. The addressing problem is simplified since each user has only one name throughout the system, rather than one for each host in the system. The availability and accessibility problems are reduced since no single host is responsible for all of a user's mail submission and retrieval. Server-based approaches, however, have so far been limited to homogeneous environments. (Systems such as Grapevine are connected with the rest of the world through gateways that act as mediators between the two models.)

The HMS Approach

The goal of the HMS is to accommodate heterogeneity, including the host-based model, while solving the problems of addressability, availability, and accessibility in the style of the server-based model. It must also be relatively easy to accommodate new mail systems as they become available. Our approach is structured similarly to that of the name service. Our mail service must coexist with existing mail services, with users permitted to continue using existing, unmodified user agents.

The HMS model consists of four layers. The top layer represents the user agents of the HMS. The second layer represents the HMS server, which is responsible for retrieving and sending mail to and from HMS user agents. The third layer represents a set of mail semantic managers (MSMs), one for each distinct integrated mail system, each of which is responsible for transforming

operations in the underlying mail systems to and from HMS server operations. The bottom layer represents the existing mail systems themselves. The HMS server and the MSMs are remote servers, allowing interaction through the HRPC facility.

The basic objective of the HMS server is to implement the server-based model across heterogeneous mail repositories, such as mail servers and local spool files. In our prototype we integrate the UNIX mail system with the Xerox Clearinghouse mail system. We could not make the HMS server specific to these two systems, however, without increasing the difficulty of integrating a new system later. So, we implemented the HMS server in terms of an abstract interface, including operations to get and deliver messages, that the individual MSMs must implement. This structure makes it easy to have the HMS server check all repositories on mail reading and any repositories on mail sending, regardless of whether the mail repositories are mail servers, files, or a mixture of the two.

In addition to the uniform interface between MSMs and the HMS server, the model requires that the HMS server provide a uniform interface to HMS user agents, so that introduction of new mail systems does not necessitate changes to the user agents. The interface is identical to that provided by MSMs, except that the operations apply to all of a user's mail repositories, rather than the specific one associated with a given MSM.

The HMS server maintains several databases that map names to information about users, services, and distribution lists. These databases are used to control mail delivery and retrieval, as well as naming translation across the different mail systems. Through careful use of indirection, we avoid reregistering information from underlying mail systems (such as alias databases) into these HMS databases. The *mailbox database* maintains mailbox information for each HMS user, each of whom might have mailboxes in multiple mail systems. The entry for each user indicates to the HMS server which repositories are associated with the user. Each user has a unique HMS name associated with the appropriate master mailbox list. Each HMS name must be unique, since each HMS user has its own master mailbox list. Since an HMS user may invoke the HMS from any local system, the HMS must provide a mapping from each user's name to the user's HMS name. This many-to-one mapping is represented in the HMS's *global alias database.*

The basic operation combines the server-based approach with the indirection of the HNS. When a user reads mail from an HMS user agent, the user's local name is mapped to his or her HMS name, using the global alias database. Using this name the list of mail repositories is retrieved from the mailbox database. The HMS server then iterates through the MSMs associated with every one of these repositories (the mapping to the MSM is included in the database), invoking the desired retrieval operation. The messages retrieved are merged together and returned to the user agent. When a user sends mail from an HMS user agent, the same mappings are done. The difference is that the HMS server looks for a single MSM to succeed in sending the message, since only one instance of the message need be sent.

Our model and prototype are related to both the Cornell Bridge system [17] and Norman system [13], but we provide richer interconnection and a structure more suitable for integrating additional mail systems. The X.400 effort [14], also called MHS (message handling system), intends to connect different systems using a particular message format and protocol suite; this contrasts with the HMS, which does not attempt to define such standards. The HMS could provide interconnection between X.400 services and other mail services by defining an MSM for X.400.

FILING

We have approached heterogeneous filing in two ways: The first effort defined a centralized file service that permits the storing and retrieval of multiple representations of the same data. The second effort is structured more like that of the naming service, where existing file systems are used as the basic medium for storing data. (In contrast to the other facilities and services, our efforts in filing have to date been more investigational than production oriented.)

Both efforts have many similar premises drawn from the needs of a heterogeneous environment. In most homogeneous file systems, the storage of data alone allows information to be shared: The file system accepts and delivers streams of raw bytes. In a heterogeneous environment, storage of data is not the same as storage of information. Consider a stream of data consisting of a sequence of records, each containing an integer, a character, and a floating-point number, written by a Mesa program on a Xerox workstation. To read these data into a Pascal program executing under VMS, transferring the bytes is not enough. It is also necessary to address the heterogeneity of the programming languages, operating systems, and underlying hardware. Language heterogeneity means that record packing and padding characteristics differ. Operating-system heterogeneity means that the file system calls used by the two programs may differ, as may the underlying file structures. Hardware heterogeneity means that the byte ordering of integers and the representation of floating-point numbers differ.

Our approach to both language and hardware heterogeneity is to rely on the HRPC facility, which accounts for the differences in data representation. We do not attempt to overcome operating-system heterogeneity. Instead, our filing service is not transparent: Client programs are required to distinguish between access to local file systems and the HCS filing service. Nontransparency, which is embraced by other file systems such as those from Xerox PARC [11, 35, 36], is necessitated by our assumption that the code of existing systems cannot be modified.

A Centralized Approach

In the first approach to filing, we viewed writing a file from one system and later reading it from another as similar to a very slow RPC. Like an RPC, a user writes a record from one machine and later reads the logically equivalent record onto another. Any necessary representational changes are automatic. For this to be possible, each file is considered to be a sequence of typed records. The record type is described, using the HRPC interface description language, when a file is created. The type is stored along with the file, and a mechanism ensures that programs reading records from the file interpret the data correctly. Just as we allow multiple data representations in HRPC, we permit files to be stored on disk in any legitimate HRPC format.

The benefits of accommodating multiple representations rather than defining a standard representation hold in filing as well as in HRPC. Besides being potentially faster, storing the RPC format avoids any loss in accuracy from unnecessary conversions. Although the prototype server is implemented using HRPC, the clients typically run their own native RPC; hence, it is the data representations of these native RPCs that are stored by the file system.

The prototype is composed of a type server, a file server, and a mechanism for generating routines for accessing the file system. The type server provides storage for the file type database, assigning a unique identifier to each distinct type. To avoid problems of circularity, we implemented a separate type server, rather than storing type information in the HCS filing service. The file server provides directory, read, and write operations. The directory operations include list, delete, and create directory. Files are immutable, permitting readers to open and read from the random access files at any time.

Reading and writing are implemented using generic routines parameterized by the type of file record, allowing accommodation of an increasing number of file types without requiring recompilation of the server. These server routines receive RPC calls from a client, demarshaling the arguments according to the file type. Although the same technique could have been used by clients, the lack of support for polymorphism in most programming languages makes it less than elegant; even naming the read and write routines for different types is clumsy. Instead, we modified the HRPC stub generator to produce type-specific client stubs, easing the creation of read and write routines specific to a given language and file type. The modified stub generator also registers new file types with the type server.

The file system is used either through standard utility programs, such as `get-text-file` and `put-text-file`, or by reading and writing individual records under program control. To write a program that uses the file system, the user first describes the file record types in Courier IDL, extended with a new keyword `FILETYPE`. This description is then compiled with our modified HRPC stub compiler, producing a set of stubs specific for reading and writing files of each of the declared types.

The implementation of the polymorphic server routines that accept type parameters was complex given HRPC's statically typed call model. We needed dynamic typing, and only a small change to the HRPC stubs, in combination with a dynamic parameter demarshaler, gave us this capability. The dynamic demarshaler is implemented as a procedure that accepts a record type and two HRPC `Bindings`. It reads bytes from the input `Binding`, breaking them up into data types according to the record type and input format, and then writes the same data to the output `Binding`, using the output format. The `Bindings` can represent any combination of RPC components; as a result it is possible to convert between any of the supported formats. If the `Bindings` have the same format, then a simple copy is done.

The Face–Finger Service used as an example under "Remote Procedure Call" is in fact based on our prototype implementation of this approach to heterogeneous filing. The data for each user are stored in a separate file. When a client wants the information, it makes calls to the filing service to open and read the appropriate file. Because of record typing, all necessary data conversions are done at the server, and the client receives the data ready to display. Of course, a client need not conform to the baseball card format. For example, an initial prototype client just displayed the text information and ignored the picture. Another client might take advantage of the tagged data and display the office and office phone number side by side, something that would not be possible if all the information were contained in one homogeneous string.

This approach to filing is similar to the File Transfer, Access and Management (FTAM) effort [23], an ISO standard that provides remote access to files. FTAM defines a large, general interface that must be provided across all participating hosts. FTAM access is available to those files stored in a special FTAM file store. To make a file available through FTAM, the user must write a type description and then place the file in the FTAM store.

A Decentralized Approach

Our first approach (and the FTAM approach) has at least two drawbacks: The shared files are separate from the conventional local files, and each file must have an associated, and often cumbersome, type description. These problems have led us to explore an alternative approach, based on a structure similar to that of HNS, that allows us to overcome these differences in the same ways that NSMs overcome differences in syntax and semantics of name services. This approach is in some sense the heterogeneous analogy to mounting pieces of a global file system in a distributed, homogeneous environment.

This effort adopts a much different approach than the first. Rather than requiring that shared files be kept

in a separate, logically centralized service, we attempt to provide sharing of all files stored in the union of the file systems available on each of the systems in the environment. Just as in naming, the advantage of this is that no explicit reregistration step is needed. Not only is this more convenient for creators of new applications, but existing software that manipulates local files only can run unaltered with its results still being available to all of the environment's systems. Because we do not impose a structure on existing files, the approach focuses on providing an access method for these files.

As demonstrated by the first prototype, HRPC can handle data representation heterogeneity if knowledge of the data types of the file contents is available. So, finding the data type of each item stored in the file then is our first problem. The type server of the first approach is unsatisfactory, since it requires significant reregistration of information, which we wish to avoid.

Since there is no stored type information available on the files stored locally, either the client must provide the type, or else the type must be inferred. In the first case, the client provides the type information when opening a file. This is an approach commonly employed in homogeneous environments, where no type checking of file contents occurs. Unfortunately, it is not reasonable to expect the client opening the file to know this information, since it may depend on the system on which the file is stored and the client is ignorant of that information. Instead, we infer the types by using sets of system-specific defaults, based on information such as file name extensions. Exceptions to these defaults can be registered explicitly; hence, the information stored is proportional to the number of exceptions rather than the number of files. A similar problem was solved in using the naming service to perform HRPC binding [37], so we are reasonably confident of the approach.

Another reason we have avoided having clients designate the file type is that we wish to provide generic file operations in our heterogeneous environment. A typical example is `compare`, which should be defined to compare two files for equality regardless of the types of files. Type inference allows us, in cases where inferencing succeeds, to construct generic programs.

The file typing problem goes even deeper in some heterogeneous environments. Whereas UNIX supports only a single, very simple file type (a stream of bytes), other file systems support multiple organizations. HRPC's support for translation does not solve this problem directly, since different organizations may require translations at both the basic data-type level as well as the organizational level.

We are in the process of implementing the prototype for this approach to filing. Once it is completed, we intend to use it as a base for continuing our research into several topics, including organizational translations, run time performance measurements, and simply determining the effectiveness of this style of shared file system.

INTEGRATING A NEW SYSTEM TYPE

Significantly reducing the cost of integrating a new system type into a computing environment is the primary goal of our work. What does this integration actually entail?

The first task in accommodating a new system is to allow it to communicate with the HRPC facility. Systems that already have a native RPC system are generally straightforward to integrate. In these cases the new system need not be modified at all. Instead, HRPC must be updated to incorporate all components of the native RPC that are not yet known by HRPC. Commonly the transport protocol will already be known by HRPC, whereas others, such as the binding and control protocols, may have to be defined.

Systems with no native RPC system require an implementation of one. Typically we would implement a subset of HRPC: one instance of each of the components. In many cases at least a suitable transport protocol will already exist and can be adapted merely by providing a thin veneer. For the other components, code that implements *any* HRPC-understandable instance can be ported to the new system. Only in situations where a component does not exist and cannot be ported to the new system must a new piece of code be written. In the worst conceivable case—a system with no native RPC, no existing components, and no possibility of porting code from another machine—the situation is no worse than the pre-HCS situation in which a full RPC system must be implemented from scratch.

In all three cases, clients and servers on the newly integrated system can talk to any clients and servers in the core HRPC system. In situations where one wishes to broaden the scope of systems with which the new system can communicate, either more instances of each RPC component must be constructed for the new system, or service-specific bridges must be constructed on some full-HRPC system.

The second task in accommodating a new system is to integrate it into the HNS. If there is a native naming service on the system, and the service is one that has already been incorporated in the HNS, then the integration is already complete. But, if the system uses a native naming service that has not yet been integrated into the HNS, the new native naming service must be registered with the HNS, and NSMs must be defined for all appropriate query types. Both of these cases permit existing clients of the native name services to continue working without change. New clients can use the newly defined HRPC support to access the HNS directly. After the HRPC and HNS facilities are supported on the new system, the remote computation, mail, and filing services must be installed.

Installing support for remote execution on a new node is largely a matter of implementing TPL interpreters for the new system and adding any needed local function. Often the implementation of TPL interpreters will be a simple port, although occasionally a more

significant effort might be required. Once the interpreters are built, existing clients and servers can be used, and new clients and servers easily created.

Integrating a new mail system requires three steps: First, MSMs for the new mail system need to be built; in our prototype, MSMs were crafted using significant chunks of code from UNIX and Xerox user agents. Second, at least one user agent that uses an HMS server for mail delivery and retrieval must be built; this can be done by porting HMS user agents to the new environment or else modifying existing user agents of the new mail system to access the HMS server. Third, the new mail system's delivery agent must be configured to use an HMS server as its delivery mechanism. This is generally a minor task since most mail systems have some configuration mechanism to specify the location of gateways servicing nonlocal mail networks; if such a change is impossible, all user agents in the new mail system must be modified to use an HMS server.

Since both approaches to filing are based on nontransparent calls, constructing new clients is relatively simple. In the first approach, the type and file servers can be used without change, but read and write stubs must be defined to construct clients on the new system. If the core HRPC system has been built for the new system, the stub generator for the file service will be similarly easy to build. If the new system has only its native RPC system, then the stubs can be generated by hand or, more likely, by modification of the stub generator for the native system. No servers need be defined for the first approach, of course, since the centralized servers are already available. To define clients, the second approach requires only the availability of HRPC. To integrate the new native filing system, the filing service equivalents of NSMs need to be defined, just as for the HNS.

CONCLUSION

Our initial interest in heterogeneity came from two directions. One was our belief that the ever-growing interconnection of diverse systems is leading to a situation in which we will be hard-pressed to easily take advantage of the broad set of resources available through this "meganet." The other was the specific problems we face every day due to heterogeneity in our local computing environment. Our work is drawing us closer to meeting our day-to-day needs. This experience is giving us insight to solutions that may apply in the broader case.

Although each of our network facilities and services was designed to meet specific goals, a number of common themes exist:

- *Emulation.* We do not integrate heterogeneous systems by defining new standards that all systems must support. Instead, we build software that can emulate relatively easily a range of existing facilities. We accomplish this emulation by factoring the design of subsystems into easily replaced parts. HRPC is the best example of emulation.
- *Localized translation.* Different systems store and interpret shared information in different ways. With many system types, centralizing the responsibility for all combinations of translations is unmanageable. Instead, we place this responsibility for translating between representations in the hands of the entities that know the most about it. One specific kind of translation—the type conversions that arise in every facility and service—is automatically managed as much as possible. The HNS's NSMs are another example of localized translation.
- *Procrastination.* We make decisions, such as those involved in binding, as late as possible. This permits us to place less specific information in the code itself, making it easier to accommodate new systems. Procrastination is facilitated by factoring, since choices about how to select an individual component can be delayed without significant modification of code.
- *Complex services and simple clients.* To allow the creation of new clients at significantly reduced costs, we must increase the sophistication of many services. By complexity we do not mean that we define more extensive and difficult-to-use interfaces to services; indeed, the interfaces must be simple to ease construction of the clients. Rather, we mean that the function performed by the service is often more complex and time consuming than might be desired in the absence of the problems posed by heterogeneity.

We believe that "heterogeneity through homogeneity," that is, defining a new standard that must be adhered to by all systems, is an approach that has serious limitations, especially in the near term. In the best of all possible worlds, standardization would allow diverse systems to communicate and to share infrastructure. There is little indication, though, that the current trend toward ever-greater diversity will reverse itself quickly; an interim solution is needed. Hence, emulation and accommodation are hallmarks of the HCS approach.

Acknowledgments. Many thanks go to the students who have participated in the HCS Project, including Brian Bershad, Fran Brunner, Dennis Ching, Sung Kwon Chung, Bjorn Freeman-Benson, Kimi Gosney, John Maloney, Cliff Neuman, Brian Pinkerton, Michael Schwartz, Mark Squillante, James Synge, and Douglas Wiebe.

REFERENCES

1. Allman, E. Sendmail—An internetwork mail router. *UNIX Programmer's Man. 4.2BSD*, 2C (Aug. 1983).
2. Almes, G.T., Black, A.P., Lazowska, E.D., and Noe, J.D. The Eden system: A technical review. *IEEE Trans. Softw. Eng. SE-11*, 1 (Jan. 1985), 43–58.

3. Bairstow, J. GM's automation protocol: Helping machines communicate. *High Technol. 6*, 10 (Oct. 1986), 38–42.
4. Balkovich, E., Lerman, S., and Parmelee, R.P. Computing in higher education: The Athena experience. *Commun. ACM 28*, 11 (Nov. 1985), 1214–1224.
5. Bershad, B.N. Load balancing with Maitre'D. Tech. Rep. UCB/CSD 86/276, Computer Science Division (EECS), Univ. of California, Berkeley, Dec. 1985.
6. Bershad, B.N., and Levy, H.M. Remote computation in a heterogeneous environment. Tech. Rep. 87-06-04, Dept. of Computer Science, Univ. of Washington, Seattle, June 1987.
7. Bershad, B.N., Ching, D.T., Lazowska, E.D., Sanislo, J., and Schwartz, M. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Trans. Softw. Eng. SE-13*, 8 (Aug. 1987), 880–894.
8. Birrell, A.D., and Nelson, B.J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1 (Feb. 1984), 39–59.
9. Birrell, A.D., Levin, R., Needham, R.M., and Schroeder, M.D. Grapevine: An exercise in distributed computing. *Commun. ACM 25*, 4 (Apr. 1982), 260–274.
10. Black, A., Lazowska, E., Levy, H., Notkin, D., Sanislo, J., and Zahorjan, J. An approach to accommodating heterogeneity. Tech. Rep. 85-10-04, Dept. of Computer Science, Univ. of Washington, Seattle, Oct. 1985.
11. Brown, M.R., Kolling, K.N., and Taft, E.A. The Alpine file system. *IEEE Trans. Comput. Syst. 3*, 4 (Nov. 1985), 261–293.
12. Brownbridge, A., Marshall, A., and Randell, A. The Newcastle connection—Or UNIXes of the world unite. *Softw. Pract. Exper. 12*, 12 (Dec. 1982), 1147–1162.
13. Callahan, J., and Weiser, M. Norman Mailer: A multiple protocol mail reading/composing program. In *Proceedings of the International Working Conference on Message Handling Systems (State of the Art and Future Directions)* (Munich, Germany, Apr.). IFIP, Arlington, Va., 1987, pp. 2.2-1–2.2-16.
14. CCITT. Recommendations X.400 to X.410. Document 66 of the 8th Plenary Assembly of the CCITT. Doc. AP VIII-66-E, CCITT, June 1984.
15. Dannenberg, R.B. Resource sharing in a network of personal computers. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., Dec. 1982.
16. Dineen, T.H., Leach, P.J., Mishkin, N.W., Pato, J.N., and Wyant, G.L. The network computing architecture and system: An environment for developing distributed applications. In *Proceedings of the USENIX Conference* (Phoenix, Ariz., June). USENIX Association, Berkeley, Calif., 1987, pp. 385–398.
17. Field, J. *The XDE/UNIX Bridge.* Cornell Univ., Ithaca, N.Y.
18. Gibbons, P.B. A stub generator for multi-language RPC in heterogeneous environments. *IEEE Trans. Softw. Eng. SE-13*, 1 (Jan. 1987), 77–87.
19. Gosney, K. Heterogeneous remote procedure call for Franz Lisp. M.S. thesis and Tech. Rep. 87-07-03, Dept. of Computer Science, Univ. of Washington, Seattle, July 1987.
20. Gray, T.E. Position paper for workshop on "Making Distributed Systems Work." In "Making Distributed Systems Work" (Amsterdam, The Netherlands, Sept.). 1986.
21. Hagmann, R. *Summoner Documentation.* Xerox PARC, Palo Alto, Calif., July 1985.
22. Hayes, R., and Schlichting, R.D. Facilitating mixed-language programming in distributed systems. *IEEE Trans. Softw. Eng. SE-13*, 12 (Dec. 1987), 1254–1264.
23. International Organization for Standardization. Information processing systems—Open systems interconnection—File transfer, access and management. Draft International Standard 8571, OMNICOM Information Service, Vienna, Va., Apr. 1985, parts 1–4.
24. Johnson, J.Q. *XNS Courier under UNIX.* Cornell Univ., Ithaca, N.Y., Mar. 1985.
25. Jones, M.B., and Rashid, R.F. Mach and Matchmaker: Kernel and language support for object-oriented distributed systems. In *Proceedings of OOPSLA 86* (Portland, Oreg., Sept.). 1986, pp. 67–77.
26. Jones, M.B., Rashid, R.F., and Thompson, M.R. Matchmaker: An interface specification language for distributed processing. In *Proceedings of the 12th ACM Symposium on Principles of Programming Language* (New Orleans, La., Jan.). ACM, New York, 1985, pp. 225–235.
27. Kaminski, M.A., Jr. Protocols for communicating in the factory. *IEEE Spectrum 23*, 4 (Apr. 1986), 56–62.
28. Kronenberg, N.P., Levy, H.M., and Strecker, W.D. VAXclusters: A closely-coupled distributed system. *ACM Trans. Comput. Syst. 4*, 2 (May 1986), 130–146.
29. Lampson, B.W. Designing a global name service. In *Proceedings of the 5th ACM Conference on Principles of Distributed Computing* (Calgary, Canada, Aug.). ACM, New York, 1986, pp. 1–10.
30. Leach, P., Levine, P.H., Douros, B.P., Hamilton, J.A., Nelson, D.L., and Stumpf, B.L. The architecture of an integrated local network. *IEEE J. Sel. Areas Commun. SAC-1*, 5 (Nov. 1983), 842–856.
31. Morris, J.H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S.H., and Smith, F.D. Andrew: A distributed personal computing environment. *Commun. ACM 29*, 3 (Mar. 1986), 184–201.
32. Notkin, D., Hutchinson, N., Sanislo, J., and Schwartz, M. Heterogeneous computing environments: Report on the ACM SIGOPS workshop on accommodating heterogeneity. *Commun. ACM 30*, 2 (Feb. 1987), 132–140.
33. Oppen, D.C., and Dalal, Y.K. The clearinghouse: A decentralized agent for locating named objects in a distributed environment. *ACM Trans. Off. Inf. Syst. 1*, 3 (July 1983), 230–253.
34. Postel, J., and Reynolds, J. Domain requirements. Rep. RFC 920, Information Sciences Institute, Univ. of Southern California, Los Angeles, Oct. 1984.
35. Schroeder, M., Gifford, D., and Needham, R. A caching file system for a programmer's workstation. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles* (Orcas Island, Wash., Dec.). ACM, New York, 1985, pp. 25–34.
36. Schroeder, M.D., Gifford, D.K., and Needham, R.M. The Cedar file system. *Commun. ACM 31*, 3 (Mar. 1988), 288–298.
37. Schwartz, M. Naming in large, heterogeneous systems. Ph.D. dissertation, Dept. of Computer Science, Univ. of Washington, Seattle, July 1987.
38. Schwartz, M., Zahorjan, J., and Notkin, D. A name service for evolving heterogeneous systems. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles* (Austin, Tex., Nov.). ACM, New York, 1987. To be published.
39. Sheltzer, A.B., Lindell, R., and Popek, G.J. Name service locality and cache design in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems* (Cambridge, Mass., May). IEEE, New York, 1986, pp. 515–522.
40. Squillante, M.S., and Notkin, D. A mail system for local, heterogeneous environments. Tech. Rep. 87-07-04, Dept. of Computer Science, Univ. of Washington, Seattle, July 1987.
41. Sun Microsystems. *Remote Procedure Call Protocol Specification.* Sun Microsystems, Jan. 1985.
42. Sun Microsystems. *External Data Representation Reference Manual.* Sun Microsystems, Jan. 1985.
43. Terry, D. Distributed name servers: Naming and caching in large distributed computing environments. Ph.D. dissertation, Computer Science Division (EECS), Univ. of California, Berkeley, Feb. 1985.
44. Terry, D., Painter, M., Riggle, D., and Zhou, S. The Berkeley internet name domain server. Tech. Rep. UCB/CSD 84/182, Computer Science Division (EECS), Univ. of California, Berkeley, May 1984.
45. Walker, B., Popek, G., English, R., Kline, C., and Thiel, G. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles* (Bretton Woods, N.H., Oct.). ACM, New York, 1983.
46. Xerox Corporation. Courier: The remote procedure call protocol. Tech. Rep. XSIS 038112, Xerox Corporation, Dec. 1981.

CR Categories and Subject Descriptors: C.2.4 [**Computer-Communication**]: Distributed Systems; H.2.5 [**Database Management**]: Heterogeneous Databases
General Terms: Design, Management
Additional Key Words and Phrases: Heterogeneous computer systems

Authors' Present Addresses: David Notkin, Edward D. Lazowska, Henry M. Levy, Jan Sanislo, and John Zahorjan, Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195; and Andrew P. Black, Digital Equipment Corporation, 550 King Street, Littleton, MA 01460.