Accommodating Heterogeneity

John Zahorjan, Edward D. Lazowska, Henry M. Levy,
David Notkin, and Jan Sanislo

Department of Computer Science
University of Washington
Seattle, Washington 98195

ABSTRACT

The Heterogeneous Computer Systems project at the University of Washington has over the past two years designed and implemented a software infrastructure to accommodate heterogeneous systems. We have addressed an environment consisting of a potentially large number of different system types but only a few instances of each type. Such an environment arises naturally in research and other settings where individual systems are obtained for the specialized services they provide. Our goal is not to mask the heterogeneous nature of the systems by imposing a standard interface on them, but rather to provide loose integration through a set of network services. In particular, we provide remote procedure call, naming, filing, remote computation, and mail services accessible from all system types.

1. Introduction

This paper describes the software infrastructure designed and implemented by the Heterogeneous Computer Systems (HCS) project at the University of Washington [Black et al. 1985, 1987]. This work has two goals:

- We hope to decrease the cost of adding a new type of system to an existing computing environment.
- We hope to increase the set of common services that users in a heterogeneous environment can expect to share.

We wish to achieve these goals under very weak assumptions about the nature of the individual systems.

In our environment – an academic department with a significant experimental research component – heterogeneity is a fact of life. Experimental computer research is often best conducted on a high-level testbed (e.g., Lisp and Smalltalk machines, multiprocessor workstations). Further, such research often produces unique hardware/software architectures (e.g., prototype distributed systems, special-purpose image analysis hardware). Our own environment consists of more than 15 significantly different hardware/software systems. Many other environments, both academic and industrial, have similar composition.

The loose interconnection that today accompanies heterogeneity poses several significant problems. One problem is *inconvenience*. An individual either must be a user of multiple systems or else must accept the consequences of isolation from various aspects of the local computing environment. A second problem is *expense*. The hardware and software of the computing environment are not effectively amortized, making it much more costly than necessary to conduct a specific effort on the system to which it is best suited. 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.

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.

These problems exist despite the widespread availability of communication protocols such as TCP. Users require a diverse set of services and applications; they also require the ability to construct new services and applications readily. The file transfer and remote terminal programs that currently are the standard ways of accessing foreign machines are insufficient, and constructing new services and applications on top of protocols such as TCP is too difficult.

In addressing these problems, the specific focus of the HCS project is heterogeneous environments with two key characteristics:

- There are a large number of system types with a small number of instances of certain of these types.
- Each system exists because its unique properties make it well-suited to some specific application.

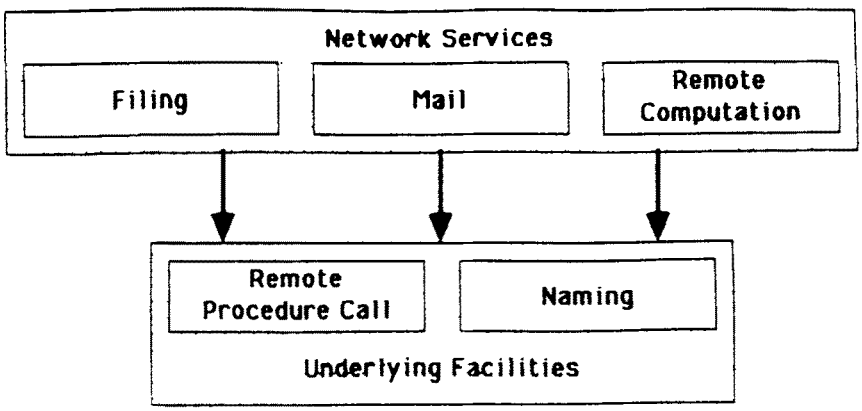

Figure 1: Relationship of HCS Facilities and Services

These characteristics, while certainly not universal, are widespread, and have significant implications: 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 us 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 and naming – support a set of key network services that are adapted to the demands of a heterogeneous environment – filing, mail, and remote computation. 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 remote procedure call and naming facilities that we also provide. The services we have selected are those that are fundamental to cooperation and sharing.
- In both the services and the underlying facilities, we attempt not to legislate standards but rather to accommodate multiple standards. This allows the integration of unmodified or minimally modified systems into our computing environment.
- We do not attempt to provide existing programs with transparent access to services. The primary reason is that we have so many system types that performing and maintaining the modifications necessary for transparency would require an effort out of proportion to the benefits obtained.
- 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.

Although each network service was designed to meet specific goals, a number of common themes exist:

- *Emulation.* We do not integrate heterogeneous systems by defining a new standard that all systems must support. Instead, we build subsystems that can relatively easily emulate a range of existing facilities. We facilitate this emulation by factoring the design of subsystems into easily replaced parts.
- *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 – are automatically managed for the user to the fullest extent possible.
- *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.
- *Complex Services, Simple Clients.* To allow the creation of new clients at significantly reduced costs, the clients must be kept simple. Thus, the bulk of the processing must be performed by the servers.

The HCS project thus has a coherent approach to dealing with a particular style of heterogeneity. There are a number of other styles of heterogeneity, each of which demands a somewhat different solution. MIT's Project Athena [Balkovich et al. 1985, Gettys 1984] and CMU's ITC Project [Morris et al. 1986] are two highly visible heterogeneity-related efforts. Each seeks to accommodate heterogeneity through *coherence*: enforcing high-level uniformity in software while permitting implementation on diverse hardware. Both projects rely primarily on UNIX[1]. Project Athena is standardizing on an applications interface, the ITC on a centralized file service. Another major effort is the MIT LCS Styx project (formerly called the Common System), 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 [Popek et al. 1981, Walker et al. 1983], 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* [Gray 1986] with the goal of integrating machines with dissimilar operating systems. Finally, GM MAP [Kaminski 1986] is a major industrial effort that involves top-to-bottom standardization in a specific application domain – manufacturing.

In the remaining sections we present an overview of each of the five services that comprise the HCS.

2. Heterogeneous Remote Procedure Call

The overall objective of the HCS project is to develop an environment in which heterogeneous computer systems share a small set of key services that provide "loose integration." Network communication is the *sine qua non* of this effort.

Although some form of networking capability is possessed by all of the systems that are likely to be of interest to us, no single protocol is shared by all of them. This absence of protocol standardization is one serious impediment to accommodating heterogeneity.

[1] UNIX is a trademark of ATT Bell Laboratories.

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. However, the absence of low-level protocol standardization 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 providing communication is *remote procedure call* (RPC) [Birrell & Nelson 1984]. RPC supports communication among application programs while relieving them of concern for data representation, transport protocol details, etc. As much as possible, an RPC facility provides a mechanism across the communication network that has the same syntax and semantics as local procedure calls within the application program's high-level language.

We have identified five major components of an RPC facility: compile-time support (including the programming language, the interface description language (IDL), and the stub generator), the bind-time protocol, and the three call-time protocols – transport, control, and data representation. Existing RPC facilities make significantly different choices in each of these areas. 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 also are difficult to modify to make such communication possible.

Inspired by DEC SRC RPC facility [Birrell et al. 1985], we have attempted to modularize the components of our HCS RPC facility (HRPC) [Bershad et al. 1987] by specifying clean interfaces among them. 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 to be used is determined dynamically at bind-time. This design meets two key objectives. First, we are able to *emulate* existing RPC facilities by providing appropriate black boxes. Second, 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.

A simple example will help to illustrate our approach. We designed a server that returns a list of the 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) [Xerox 1981], on Sun computers using the standard Sun RPC with UDP datagrams (i.e., the UDP protocol for transport, the XDR data representation standard, and the Sun protocols for binding and control) [Sun 1985a, 1985b], 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 can communicate with each server using that server's own native RPC; it is possible to make a sequence of calls to different servers, each call emulating a different native RPC. Performance is comparable to that of the native RPC facilities.

The Call-Time Organization of HRPC.

The basis of the HRPC factorization is an abstract model of how *any* RPC facility works, expressed through the call-time components. To isolate the actual implementation of each of these components, we define a procedural interface to each. These interfaces allow an HRPC stub and any combination of control protocol, data representation, and transport protocol components to function together. In addition to location information, a data structure called a **Binding** explicitly represents the choices for these three components as separate sets of procedure pointers. Calls to the component routines are made indirectly via these procedure pointers. By filling in a

Binding with different values for these procedure pointers, a single HRPC program can communicate with a wide variety of conventional RPC programs.

The Bind-Time Organization of HRPC.

The first step in binding is *naming*. For binding, naming is 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. Because of this, each server is typically allocated its own communications *port*, which, together with the network address, uniquely identifies the server.

Consider the case of an HRPC client importing a server written using some existing conventional RPC. The client specifies a two-part string name containing the type (e.g., "FileService") and instance (i.e, a host name) of the service it wishes to import. To honor this request, the HRPC binding subsystem first queries the HCS Name Service (described in the next section), 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 **Binding Descriptor**. To do this, the procedure pointer parts of the **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.

The HRPC system uses a stub generator for an extended version of the Courier IDL [Xerox 1981], based on the generator written at Cornell [Johnson 1985]. The "code generator" portion of the stub generator was modified to support the HRPC interface. The stub routines are generated in the C programming language. Our stub generator supports additions to the Courier IDL, the most important of which is an escape mechanism known as a **USERPROC** with which users can provide their own marshalling routines for complicated data types, such as those containing pointer references.

Evaluation.

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 to define interfaces between the various RPC components, making 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 degree of modularity necessary in HRPC might significantly increase execution time; however, initial benchmarks show that HRPC is competitive with the native RPC facilities being emulated. In Table 1 we present the results of two benchmark programs. The first, called NULL, measures the elapsed time of a parameterless remote call and return. The second, called BYTES, measures the elapsed time to transfer a string containing 37,000 characters. For each of these benchmark programs we compare the performance of the Sun RPC facility to that of HRPC emulating the Sun RPC, and we compare the performance of the Xerox Courier RPC facility to that of HRPC emulating Courier. Times are expressed in milliseconds. The performance of HRPC is essentially identical to the performance of the native RPC facility in each case. Recall,

though, that for each benchmark, the HRPC client that calls the Sun server is *exactly the same executable file* as the HRPC client that calls the Courier server.

RPC Facility		Benchmark	
Client	Server	NULL	BYTES
Courier	Courier	22	5243
HRPC	Courier	22	4825
Sun	Sun	27	625
HRPC	Sun	28	620

Table 1 - Comparison of HRPC and Native RPC Call Times (msec.)

3. Naming

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

Two primary goals must be met by a name service for our environment. First, existing applications on the individual subsystems of the HCS should continue to run unaltered. Second, new applications written to use our global service should have access to the naming information contained in new subsystems attached to the HCS without requiring recompilation or relinking.

In creating a name service that meets these goals, there are three significant problems that arise due to heterogeneity. The first concerns the syntax of names. Because the separate subsystems of the HCS are likely to have conflicting name syntaxes, it is not possible to impose a single syntax for the global name space that would be "natural" on all systems. The global name space should make the minimum possible demands on users to adjust to an unfamiliar syntax.

The second problem is that of the *name conflicts* that occur when two or more systems containing an identical name are combined. For instance, both the Xerox and UNIX systems might contain the name **Notkin** and store data on the mail destination for that user. While this name is unambiguous when issued in an environment consisting of only one of the two systems, it is ambiguous when they are combined.

The final problem is the difficulty of simultaneously meeting the two goals outlined above. Continued execution of existing applications requires that names be accessible in the existing name services local to the individual subsystems. 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 the global service. However, such copying carries with it the difficult problem of maintaining consistency between the local and global copies.

We have constructed a global name service, the HNS [Schwartz 1987, Schwartz et al. 1987], to address these problems. Primarily because of the consistency problems just noted, we have chosen not to perform reregistration in the HNS, but to use the local name services directly to store the data associated with the global name space. The problems implied by this are discussed in the next section, which describes the design of the HNS.

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, that data is maintained in an existing name service, where it is 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 identifies all or part of 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 is associated in that service. While the HNS does not impose any restrictions on the syntax of the 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 approach for constructing HNS names ensures that these 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. Since each local name service guarantees the lack of conflicts of its local names, and since the individual name to local name mapping is invertible, no single name service can create a conflict within a context.

While the name syntax and data management scheme guarantee the absence of naming conflicts, they create another significant problem as a side-effect. This problem is that the local name services may store equivalent data in different formats or may store similar but not identical information of a particular type.

Ideally the HNS should insulate the client program from this *semantic heterogeneity*. In particular, it would be inconvenient to require every client program accessing the HNS to understand the semantics of each underlying name service and to be recompiled whenever a new subsystem type was added to the environment. Instead, the HNS should provide a single data representation and semantics that are invariant under the addition of new subsystems.

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 HNS query type is supported by a set of NSMs with identical interfaces, with one NSM for each local name service. (Figure 2 contains a diagram of the logical structure of the HNS.) Because they are remote procedures, NSMs can be added to the system while it is running without recompilation of any existing code. When a new subsystem is added to the HNS, we construct NSMs for that subsystem, register these NSMs with the HNS, and the data stored in them becomes available through the HNS.

HNS Name Lookup Procedure.

Figure 2 shows the logical structure of the HNS and the process followed in satisfying an HNS request. There are two phases. In the first, 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 that NSM to the client.

In the second phase of service, 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

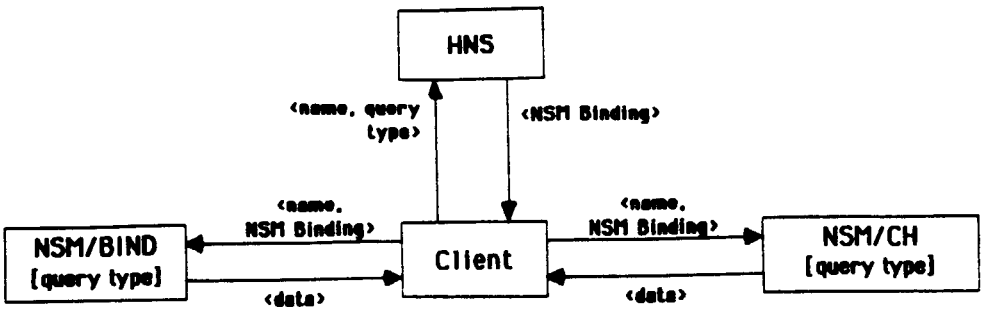

Figure 2 - HNS query processing

looking in a local file, may be employed). This information is then returned to the client.

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

While the above procedure indicates the logical process followed in satisfying HNS requests, an implementation need not be structured in an exactly analogous 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, the 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 some other 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.

Prototype Implementation.

A prototype HNS implementation exists and supports a limited number of query types, including HRPC binding, on the BIND [Terry et al. 1984, Zhou 1984] and Clearinghouse [Oppen & Dalal 1983] name services. Table 2 presents measurements of the performance of the HNS as a function of various assumptions about caching effectiveness and colocation. Each row of the table represents a different way of linking the client, HNS, and NSM components together. Components within brackets are linked locally to each other, and so can communicate using local procedure call. Components separated by brackets are remote and must communicate using HRPC. The different colocation schemes represent a tradeoff in efficiency against ease of management: tighter binding implies better performance but more difficult software management problems when one of the components has to change.

The two columns of the table represent the performance at the extremes of no cache hits (equivalent to no caching being performed, since cache lookup time is very fast) and perfect cache hit. In the latter case both the HNS and the NSM are assumed to have caches, the former storing NSM binding data and the latter query specific data.

The table shows that despite the additional levels of indirection imposed by the HNS, the performance can be quite good. For the sake of comparison, a single lookup to BIND costs roughly 27 ms., while a lookup to the Clearinghouse costs about 156 ms. (BIND achieves this efficiency by keeping its data in main memory at all times, and by avoiding some expensive procedures provided by the Clearinghouse, such as authentication.) The table also points out that caching is far more important to performance than colocation. Measurements of cache

Colocation	Cache Miss	Cache Hit
[Client,HNS,NSMs]	460	104
[Client][HNS,NSMs]	517	137
[HNS][Client,NSMs]	515	140
[NSMs][Client,HNS]	509	147
[Client][HNS][NSMs]	547	181

Table 2 - Performance of HRPC Binding (msec. elapsed time)
[] indicates colocation

hit ratios actually achieved in practice remain to be performed.

4. Filing

In most homogeneous file systems, the storage of *data* alone will allow information to be shared: the file system accepts a stream of raw bytes and later delivers an identical stream. In a heterogeneous environment, storage of data is not the same thing 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 this data into a Pascal program executing under VMS, transferring the bytes in not enough. It is also necessary to address the heterogeneity of the programming languages, the operating systems, and the underlying hardware. Language heterogeneity means that the 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.

There have been two distinct heterogeneous file systems built as part of the HCS project. The first, described more fully in [Black et al. 1987], provides an entirely new repository for those files to be made available network wide. This new repository provides features not available in some on some of the native systems, such as file versioning and immutability. This file system is built of three components: a Type server, a File server, and a mechanism for generating routines for accessing the files. Each file is kept in the format in which its first version was received. The Type server keeps the IDL description of the records of the file. The File server contains three logical modules: directory, reader, and writer. The last two are used to access the file, and are generic routines parameterized by the type of the file record. The final component is a modification of the HRPC stub generator to register new file types with the Type server and to produce type-specific client stubs. These special stubs are required because of the lack of support for polymorphism in most programming languages.

In this design files and their type information must be explicitly registered to be made available to other systems. Once registered, both whole file copy and per record read and write are available. A prototype implementation of this design has been completed and is in use for a number of applications.

The second file system is designed along the lines of the HNS. It differs from the first design in that no registration of files need take place: all files available on the native file systems are available network wide (subject to authorization constraints). This is accomplished by having an agent on each system that serves as an intermediary between the client and the file the client wishes to access. The HNS is used by the client to locate this agent, and HRPC is used to communicate with it. The agent provides file read and write on a record basis.

A major difficulty in this design is the lack of the type information needed to convert data formats from one system to another. Unlike the first design, there is no type information explicitly registered for the files to which access is provided. Instead, the client is required to provide a description of the record format at file open. This is generally reasonable since even in homogeneous systems the contents of a file cannot ordinarily be used sensibly without prior knowledge of its format. The one significant function apparently lost by this design is the ability to support generic functions, such as *compare*. In comparison to the first design, we have traded the ability to operate more conveniently on a much larger set of files for the inability to perform some useful generic operations.

A prototype implementation of this second design is still underway. Our goal is to make both systems available to our user community on a reliable basis to gain some feel for which approach is most useful in practice.

5. Mail

The goal of the HCS mail system (HMS) is to accommodate heterogeneity, including the host-based model (characterized by the UNIX and VMS mail services), while solving the problems of addressability, availability, and accessibility in the style of the server-based model (characterized by the Clearinghouse service). Further, it should be relatively easy to accommodate new mail systems as they become available.

Our model of a mail service consists of five components. *Submission* of mail occurs when a user directs the service to send mail to one or more other users. *Name resolution* of a submitted message occurs when a program decodes a symbolic address into a network address for use in forwarding or delivery. *Transfer* occurs when a transport protocol is used to move the message to the next (intermediate or final) destination. Name resolution is often responsible for selecting an appropriate transport protocol. Repeated name resolution, combined with the associated transfers, provides the function of routing. *Delivery* occurs when the transport mechanism delivers the message to its final destination. *Receipt* occurs when a program, acting for the recipient, takes the message from the depository and hands it to the user for reading and perhaps archiving.

Submission and receipt of mail are usually the responsibility of a program called the *user agent*. Name resolution, transfer, and delivery are usually the responsibility of a set of agents that comprise a mail transfer system.

The HMS Approach.

Our basic approach is to use a server-based structure that employs the HNS for naming and HRPC for communicating between mail servers and user agents. A basic principle is that the HMS will co-exist with existing mail services. Further, the existence of the HMS is transparent to users that continue to use existing, unmodified user agents.

The basic structure of the HMS is to construct generic submit and receive functions that mask as much heterogeneity as possible from the mail agents. These generic functions act like a query type in the HNS, providing a single interface for all user agents. For sending mail to and receiving mail from the outside world, we rely on standard protocols. Hence, we focus on these functions for only the systems included in the HMS environment.

To support these functions we define a *global alias* service that contains entries for each HMS user. These entries are similar to entries in Grapevine that indicate the set of mail servers on which the user receives mail. However, they can include not only HMS mail servers in the list but also simple host addresses that designate machines on which the mail is to be delivered, usually into a well-known file name such as **/usr/spool/mail/notkin**. The alias service also stores information about the delivery protocol to be used. The

alias service is constructed using the HNS.

When a user submits a mail message through a user agent we look up the addressee in the alias server. We omit this step if we can tell from the syntactic structure of the name that it is a name for an external site. Then, proceeding as in Grapevine [Birrell et al. 1982], we step through the list of elements returned for the addressee. Depending on the type of the element (mail server or host address), we send the message using the appropriate protocol, which was determined by querying the alias server. As soon as the message is successfully sent to any mail server or host, the process is terminated. For messages with multiple addressees we condense the actual outgoing messages appropriately.

When an HMS user attempts to read mail through an HMS user agent, the lookup is done in the alias server for the user doing the reading. Each element in the returned list is then searched through the use of the appropriate protocol. In the receipt case, all elements must be searched to ensure that all mail that was delivered to any element is received.

This structure allows a UNIX user, for instance, to designate in the alias server that mail is to be retrieved from several UNIX machines as well as from a set of Xerox Clearinghouse mail servers. Similarly, mail can be injected into HMS from any appropriately modified user agent; the benefits are that the host address of the recipient need not be known and that if the primary receiving location is unavailable, mail will be delivered elsewhere for later retrieval.

Our current prototype supports retrieval of mail to the Xerox environment from the UNIX world. This is similar in function to Cornell's Bridge system [Field], but the structure is intended to be more suitable for extension to other mail systems that could be added later. The global alias service is nearing completion; we later intend to modify **sendmail** to use the service.

6. 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 is one means by which these resources can be accessed. An important issue is thus the ease with which remote computation can be achieved to make such resources available to network users.

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 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; these must be communicated between the client and server.

Another problem typical of any heterogeneous communication is translation of data, as has been described earlier. Translation may occur on the client side, on the server side, or on both sides. A typical solution is to standardize on-the-wire formats for data. This 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.

THERE.

We have designed and prototyped a remote computation system called *THERE*: The HCS Environment for Remote Execution [Bershad & Levy 1987]. *THERE* is a facility for constructing remote execution clients and servers in a heterogeneous network; it is similar in many ways to Maitre'D [Bershad 1985] and Summoner [Hagmann 1985]. The goal of this system is to simplify the addition of new network services and to aid 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 3. 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 obvious knowledge of which role is being played.

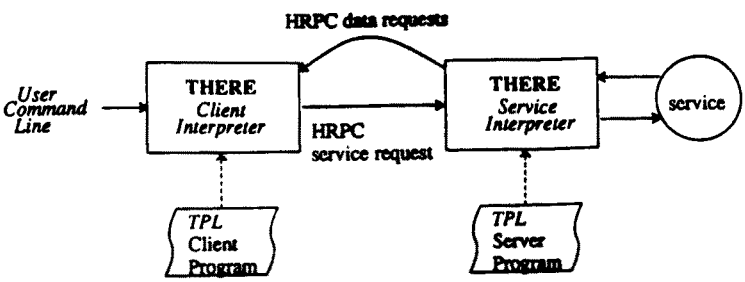

Figure 3 - Structure of THERE

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 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 post-process 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.

To date, *THERE* has been used to build servers for printer access, text formatters (TeX and troff), and remote compilers for C and ADA. The client and server agents exist in full form under 4.2/4.3 BSD UNIX, consisting of about 9000 lines of code written in C++. In contrast, the most complex server (the remote ADA compiler) has only about 200 lines of TPL code on each of the client and server.

Under UNIX, *THERE* runs on all VAX hardware, SUN's and IBM RT/PC's. A partial *THERE* client agent has been implemented in Mesa on the Xerox Dandelion workstation. Work is currently underway to complete a *THERE* port to the Tektronix Pegasus machine running UNIFLEX, a weak derivative of UNIX. Performance measurements have shown that the time spent interpreting TPL programs is negligible when compared to the time spent transferring files and executing the tasks that comprise the remote service.

7. 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.

Efforts in HCS are continuing. First, we are completing our prototypes in several the areas, including filing, mail, and remote computation. Second, we are improving the initial prototypes of other areas, especially HRPC. Third, we are broadening the number of heterogeneous systems on which our facilities and services run.

Fourth, we are defining applications that exercise our prototypes, with the dual goals of evaluating our work and improving the departmental computing facilities.

Acknowledgments. Many thanks go to the other members of the HCS project, including Brian Bershad, Jon Bertoni, Andrew P. Black, Fran Brunner, Dennis Ching, Bjorn Freeman-Benson, Kimi Gosney, John Maloney, Cliff Neumann, Brian Pinkerton, Michael Schwartz, Mark Squillante, James Synge, and Douglas Wiebe.

References

[Balkovich et al. 1985]
E. Balkovich, S. Lerman, and R. P. Parmelee. Computing in Higher Education: The Athena Experience. *Comm. of the ACM 28*,11 (Nov. 1985).

[Bershad 1985]
B. N. Bershad. Load Balancing With Maitre'D. Technical Report UCB/CSD 86/276, Comp. Sci. Div. (EECS), Univ. of Calif., Berkeley (Dec. 1985).

[Bershad et al. 1987]
Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Trans. on Softw. Eng. SE-13*, 8 (Aug. 1987).

[Bershad & Levy 1987]
Brian N. Bershad and Henry M. Levy. Remote Computation in a Heterogeneous Environment. Technical Report 87-06-04, Dept. of Comp. Sci., Univ. of Wash. (June 1987).

[Birrell et al. 1982]
A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An Exercise in Distributed Computing. *Comm. of the ACM 25*,4 (Apr. 1982).

[Birrell et al. 1985]
Andrew D. Birrell, Eric C. Cooper, and Edward D. Lazowska. SRC Remote Procedure Calls. Digital Equipment Corporation Systems Research Center (Jun. 1985). Unpublished specification.

[Birrell & Nelson 1984]
A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Comp. Sys. 2*,1 (Feb. 1984).

[Black et al. 1985]
A. Black, E. Lazowska, H. Levy, D. Notkin, J. Sanislo, and J. Zahorjan. An Approach to Accommodating Heterogeneity. Technical Report 85-10-04, Dept. of Comp. Sci., Univ. of Wash. (Oct. 1985).

[Black et al. 1987]
A. Black, E. Lazowska, H. Levy, D. Notkin, J. Sanislo, and J. Zahorjan. Interconnecting Heterogeneous Computer Systems. Technical Report 87-01-02, Dept. of Comp. Sci., Univ. of Wash. (Jan. 1987).

[Field]
J. Field. The XDE/UNIX Bridge. Cornell Univ.

[Gettys 1984]
J. Gettys. Project Athena. *Proc. USENIX Summer Conf.* (Jun. 1984).

[Gray 1986]
T. E. Gray. Position Paper for Workshop on *Making Distr. Syst. Work.* (Jul. 1986).

[Hagmann 1985]
R. Hagmann. Summoner Documentation. Xerox PARC (Jul. 1985).

[Johnson 1985]
J.Q. Johnson. XNS Courier under UNIX. Cornell Univ., (Mar. 1985).

[Kaminski 1986]
M. A. Kaminski, Jr. Protocols for Communicating in the Factory. *IEEE Spectrum* (Apr. 1986).

[Morris et al. 1986]
J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith. Andrew: A Distributed Personal Computing Environment. *Comm. of the ACM 29*,3 (Mar. 1986).

[Oppen & Dalal 1983]
Derek C. Oppen and Yogen K. Dalal. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. *ACM Trans. on Off. Inf. Systems 1*, 3 (Jul. 1983).

[Popek et al. 1981]
G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. *Proc. of the 8th Symp. on Oper. Sys. Princ.* (Dec. 1981).

[Schwartz 1987]
M. Schwartz. Naming Services in Large, Distributed Computer Systems. Ph.D. Thesis, Dept. of Comp. Sci., Univ. of Washington (Aug. 1987).

[Schwartz et al. 1987]
M. Schwartz, J. Zahorjan, and D. Notkin. A Name Service for Evolving Heterogeneous Systems. To appear *Proc. of the 11th Symp. on Oper. Sys.* (Nov. 1987).

[Sun 1985a]
Sun Microsystems. *Remote Procedure Call Protocol Specification.* Sun Microsystems, Inc., (Jan. 1985).

[Sun 1985b]
Sun Microsystems. *External Data Representation Reference Manual.* Sun Microsystems, Inc., (Jan. 1985).

[Terry et al. 1984]
D. Terry, M. Painter, D. Riggle, and S. Zhou. The Berkeley Internet Name Domain Server. Technical Report UCB/CSD 84/182, Comp. Sci. Div. (EECS), Univ. of Calif., Berkeley (May 1984).

[Walker et al. 1983]
B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS Distributed Operating System. *Proc. 9th ACM Symp. on Oper. Sys. Princ.* (Oct. 1983).

[Xerox 1981]
Xerox Corporation. Courier: The Remote Procedure Call Protocol. Technical Report XSIS 038112, Xerox Corporation (Dec. 1981).

[Zhou 1984]
S. Zhou. The Design and Implementation of the Berkeley Internet Name Domain (BIND) Servers. Report UCB/CSD 84/177, Univ. of Calif., Berkeley (May 1984).