Interconnecting Heterogeneous Computer Systems

Andrew P. Black and Edward D. Lazowska

Department of Computer Science
University of Washington
Seattle, Washington (USA)

1. Introduction

The Computer Science Department of the University of Washington has recently launched a significant research and development project whose goal is to simplify the accommodation of heterogeneity. In almost every advanced computing environment, the proliferation of different machines and operating systems is proving to be a major obstacle to the efficient use of networks of personal computers and mainframes. The current state of the art is that either expensive resources are duplicated because it is too hard to access an instance of the resource on a foreign machine, or professionals who could more profitably be concerned with other things must spend their time throwing together hacks that enable specific dissimilar machines to communicate.

In our view, the solution to this problem is to design an infrastructure whose specific goal is to radically decrease the marginal cost of adding a new type of machine to an existing computing environment, and at the same time to increase the set of common services that its users can expect to share. And this must be done under very weak assumptions about the nature of the new machine. We aim to demonstrate the feasibility of this solution by producing detailed designs and prototypes, and by testing them in our own research laboratory, which currently contains roughly twenty different types of computer. We would also like to have the opportunity to test our prototypes in other research and development environments; we believe that the problems of heterogeneity are common to universities, research laboratories and industrial system projects.

2. The Problem of Heterogeneity

In almost any state-of-the-art development or research laboratory, or any production facility where computers are used, heterogeneity is a fact of life. There are three reasons for this diversity of systems. First and most obviously, the state of computing technology is advancing rapidly: it is unlikely that the machine that was a cost-effective acquisition two years ago will still be the best buy today. But the two-year-old equipment is too valuable to be discarded; the result is that the old and the new systems must co-exist. Note that the value of the old equipment is not just the cost of replacing it, but also the two years of work that have been built upon it. The second reason is that high-level support for specific models of computation is a necessary aid to research and development involving those models; certain projects are more appropriately conducted on Lisp machines, Smalltalk machines or an integrated distributed system than on UNIX. Finally, experimental research and development projects often have as their goals the production of unique hardware/software architectures; taking an example from within our own department, the Eden experimental distributed system [1] and the Pringle, a 64 processor prototype of a massively parallel machine [5], were designed and built precisely because they are different from all our other systems.

This work is supported in part by the National Science Foundation under Grant Number DCR-8420945, by the Xerox Corporation University Grants Program, and by the Digital Equipment Corporation External Research Program.

Authors' Address: Department of Computer Science FR-35, University of Washington, Seattle, WA 98195, USA. Electronic mail: <surname>@CS.Washington.EDU or <surname>@uw-beaver.UUCP

It is true that in production and teaching environments, it is sometimes reasonable to try to make all systems appear alike, even when built from diverse hardware. However, even when this is possible, the cost may be excessive. In a research and development environment, heterogeneity is an inevitable consequence of attempting to pursue diverse project goals. It is quite inappropriate to attempt to standardise hardware, operating system, programming language or user interface.

What difficulties arise as a result of this heterogeneity, and why are these difficulties not eliminated by the almost universal availability of local-area networks running protocols such as TCP? The fundamental problem is that users require the existence of a diverse set of services and applications, and the ability to construct new services and applications readily. The file transfer and remote terminal programs that currently are the standard ways of using the network to access foreign machines are insufficient; constructing new services and applications on top of ISO Transport layer protocols such as TCP is too difficult. This low degree of logical integration has three implications:

- *Inconvenience.*

  An individual either must be a user of multiple subsystems, or else must accept the consequences of isolation from various aspects of the local computing environment. Since the consequences of isolation are unacceptable, many computer scientists regularly use two and three systems, through crude techniques such as multiple terminals/workstations and Telnet/FTP.

- *Expense.*

  The second implication is that the hardware and software infrastructure of the computing environment (file access, mail, printing, remote computation) is not effectively amortised, making it much more costly than necessary to conduct a specific research project on the system to which it is best suited. One must acquire not only the subsystem directly required to support the project in hand, but also the peripheral hardware and software necessary to make that subsystem function as a largely independent entity.

- *Ineffective use of personnel.*

  The third implication is that on those projects that require internal heterogeneity (e.g., image understanding, in which the image processing function and the inference function may be best suited to different systems), substantial effort must be diverted from the work at hand in order to address the problem of accommodating heterogeneity. Time-consuming hacks by scientists and engineers who should be doing other things are the rule, rather than the exception.

3. Accommodating Heterogeneity

Our programme of research and advanced development addresses the problem of accommodating this necessary heterogeneity – taking a collection of independent and dissimilar systems and connecting them together into a functioning whole.

Our approach is based on four key services: filing, mail, printing, and remote computation. These in turn rely on three underlying facilities: a remote procedure call mechanism, naming and binding services, and an authentication service. Our intent in the development effort is to provide a set of useful, achievable facilities while avoiding the effort required to provide facilities of decreasing marginal benefit. The fact that our model of a heterogeneous environment includes a large number of system types but a small number of instances of many of these system types has

had a pervasive influence on our design work to date, and serves to differentiate our work from that undertaken in the context of large-scale educational computing projects such as MIT's Athena [4], Brown's ISIS, and CMU's ITC [7].

- We typically do not attempt to provide existing programs with transparent access to services. Performing and maintaining the operating system modifications necessary to do so would require an effort out of proportion to the benefits obtained. Transparency can be provided for specific system types when sufficient use warrants the investment.
- Our designs tend to locate work on the servers themselves, rather than in clerk modules placed on the clients. While this tends to complicate the servers, it eliminates the need to implement complex clerks on many different types of system.
- Our desire to be compatible with existing software to the greatest possible extent causes us, in some cases, to support multiple standards rather than a single one. We feel that the benefits of compatibility will outweigh the inconvenience.

The following sub-sections summarise our plan of work, which is based on the provision of a number of services. More information, including a relatively detailed treatment of each service and a discussion of alternative approaches, will be found in reference 2. Since that report was written much progress has already been made in some areas, such as Remote Procedure call; in other areas we have not gone much further than initial design.

3.1. Remote Procedure Call

The use of Remote Procedure Call as our basic communication mechanism means that programmers can construct services based on a familiar and convenient semantic model, and that these services are isolated from the heterogeneity that inevitably will exist in areas such as data representation and transport protocol.

Our objective is not to design a new standard that must be adhered to by all of the systems in our environment. Rather, our objective an RPC facility that can be implemented on a core set of systems with maximal use of existing software. This facility will allow the core set of systems to communicate with peripheral systems by emulating the native RPC facilities of those systems.

A prototype of our Heterogeneous Remote Procedure Call facility is now functioning [3]. What we have attempted to do is to specify extremely clean interfaces between the five principal components of an RPC facility:

- the *stubs*, which are interposed between the client (also the server) and the runtime support;
- the *binding protocol*, which allows a client to locate a particular server;
- the *data representation protocol*, which determines how data is represented on the wire;
- the *transport protocol*, which determines how data is carried from one host to another;
- the *control protocol*, used internally by the RPC facility to track the state of a call.

An RPC client and its associated stub can view each of the remaining four components as a black box. These black boxes can be mixed and matched. The set of protocols to be used in communicating with a particular server is determined dynamically at bind time – long after the client has been written, the stub has been generated, and the two have been linked. This design meets the objectives stated above:

- First, we are able to emulate existing remote procedure call facilities by providing appropriate black boxes. This allows us to communicate with systems we cannot (or do not wish to) modify. This is a capability that is unique to our RPC facility.

- Second, we are able to employ existing software (e.g., transport protocols) to the maximum extent in building RPC for a new system that does not have a native RPC. Other attempts at heterogeneous RPC facilities allow this, but with much less breadth that our design.

To illustrate the potential of this approach, here is a description of a simple demonstration that we have built. We designed a server that, when called, replies with a list of the users logged in on the machine on which it resides. We implemented this server on three different systems: on Xerox computers using the standard Xerox RPC facility [9], on SUN computers using the standard SUN RPC facility [8] with UDP datagrams as the transport mechanism (one of two alternatives allowed by the SUN RPC), and on VAX computers using the standard SUN RPC facility with TCP as the transport mechanism. We then implemented a client of this service in our Heterogeneous RPC world. This single client is able to communicate transparently with each server using that server's own native RPC; it is possible to make a sequence of calls to different servers, each call employing a different selection from among the black boxes. Performance is comparable to that of the native RPC facilities.

We are utilising our prototype to test the functionality and performance of the interfaces that we have designed between the various components of the RPC facility. Incorporation of additional alternatives for each component is an important part of this testing. Our emphasis thus far has been on binding, data representation, transport, and control. Language heterogeneity has received relatively little attention. At present, our stub generator compiles Courier interface descriptions into C stubs. One of our objectives is to add Lisp as a second supported language.

3.2. Naming and Binding

A uniform, global name space is essential to effective use of any distributed computer system. Such names spaces are typically supported by a naming service, designed specifically for this function. While a single naming service is adequate for a homogeneous environment (for instance, Clearinghouse [6] is designed to accommodate the possible interconnection of all of its instances), a heterogeneous environment must allow connection of distinct naming services in a graceful way. One approach to this problem is to choose a specific name service as the network standard. This requires re-registration of names as new name services are introduced, and more importantly, the adaptation of existing applications to use the standard name service. Our approach is to provide a global name service by making use of the existing services, thus avoiding both these drawbacks. The Global Name Service (GNS) responds to name inquiries by determining the native name service in which the name exists, and directing the inquiry to that service. Thus, existing applications that create new names with a native name service can continue to run unaltered; these new names become available immediately through the Global Name Service.

It may seem that the task of the GNS is very simple. In fact, because the underlying name services may be very different, the task of providing a uniform interface to the users of GNS may be quite complicated. The structure of the GNS, which incorporates a set of distributed name semantic managers, reflects the truly heterogeneous facility it provides.

Because the name service provides the basic operation of mapping names into data, the binding service necessary for RPC can be provided as a natural function. A server registers its name and associated binding information, and a client retrieves the binding information at some later time by providing the name service with the server name. Because binding is a time-critical operation, this aspect of the naming service is more tightly integrated into the GNS than other name service functions. Our design for the GNS allows tradeoffs between efficiency and ease of implementation, which are used to advantage in the design of the binding service.

3.3. Authentication

Protection in computer systems involves two problems.

- *Authentication* is the means by which it is established that some entity within the system is in fact who or what it claims to be.
- *Authorisation* is the process of determining that some entity has the right to perform a particular operation.

Authentication is a pre-requisite to authorisation: unless a server can establish the identity of the entity requesting service, no authorisation checks can be performed. In our design, authorisation is the prerogative of individual servers; for example, the file system maintains its own access control lists, and decides which users should be allowed what kinds of access to which files. However, in order to do this, the file service must have some way of authenticating the user's identity; in essence, this is the rôle of the authentication service.

Initially, authentication will be implemented as an adjunct to the naming service. Each user of network services will be registered with the authentication service. The registration information will consist of the user's name and an encrypted version of a password known only to the user. The user will establish an authentication context by executing a network *login* utility prior to using any services. The authentication context will be passed as an explicit RPC parameter when required for access to services.

Any authentication scheme has a cost; in general, the more thorough the precautions taken against infiltration (e.g., by using increasingly sophisticated encryption techniques), the higher the cost (e.g., in lost performance). In our own (academic) environment, security is not a major issue, and so we have chosen a relatively weak approach to authentication. We are oriented towards research, which requires participation rather than exclusion; one effect of this orientation is that we are a close knit group of knowledgeable, mutually trusting users.

3.4. Filing

The sharing of information through files is an important aspect of any computing environment, and one that has become harder to support as workstations have replaced timeshared machines. We will facilitate file sharing by building the Washington Common Store (WCS), a common file storage facility that provides reliable long term storage in a heterogeneous environment. We believe that heterogeneity affects the uses to which a file service will be put, and thus the facilities that it should offer, in quite basic ways. Accommodating heterogeneity does not mean simply allowing one machine to access a file system that was designed to support a foreign operating system on different hardware. It means providing a lingua franca, a set of common storage facilities that help the diverse systems to share information, and thus to communicate.

Consider the use of a common file system by heterogeneous machines. Recall that the motivation for the introduction of a file service was to provide common storage: a repository in which information can be kept until it is needed. In many homogeneous file systems, all that is needed to accomplish this function is *data* storage: the file system accepts a stream of raw bytes, and guarantees to deliver a byte-wise identical stream when later requested. In a heterogeneous system, 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 a large integer, a character and a floating-point number, written by a Mesa program on a Xerox workstation. We would like to read this data into a Pascal language program executing on a VAX workstation. There are three levels of heterogeneity with which we must deal. Operating system heterogeneity means not only that the file system calls used by the two programs will be quite different, but that the underlying file structures may differ too. Hardware heterogeneity means that the byte-ordering of integers and the representation of floating-point numbers differ. Language heterogeneity means that the record

packing and padding characteristics differ.

The problems of language heterogeneity and hardware heterogeneity are avoided in the Washington Common Store by providing typed files. Each file is considered to be a sequence of records of arbitrary user-defined type. The type of the records is determined when a file is created, and a type-checking mechanism ensures that programs that read records from the file will not misinterpret the data.

Apart from the typing of files, the design of WCS has been adapted to accommodate heterogeneity in several ways:

- When deciding how to partition function between clients and servers, we wish to arrange that a client can access WCS with only minimal local software support, in addition to the basic RPC mechanism. We wish to avoid a requirement for extensive local caching and iterative queries from the local host. As a result, we intend to concentrate functionality in the server. The cost of doing this is greater loading on the server and a potential increase in network traffic. The benefit is simplicity of implementation: the less function provided by client code, the easier it is to port that code to new types of client machine.
- The interface to WCS does not attempt to simulate that of the client's native operating system. Rather, we expect the users of WCS to realise that they are accessing a remote resource, and to expect to use utilities that differ from those applicable to local files. The cost is non-uniformity of the user interface on a given machine. The benefits are uniformity of the WCS interface across different machines and operating systems, and significantly reduced implementation effort. These considerations are discussed further in the next section.
- WCS provides archival storage. It is not intended that a workstation should access WCS whenever it needs to load an executable binary, nor that WCS be a repository for a compiler's temporary files (except in the case where different passes of the compiler execute on different machines). Rather, we expect WCS to be the medium of choice for completed papers, various versions of a source program, documentation, and experimental data. A natural consequence of this intention is that WCS should be capable of retaining and distinguishing between multiple versions of a file, and of storing such versions space-efficiently. In addition, we have determined that versions of a file should be immutable: once written to WCS, they cannot be changed. Immutability avoids the problem of two clients trying to update a shared file simultaneously.
- The ultimate capacity of an archival common file system is difficult or impossible to predict in advance. One way of dealing with this uncertainly is to allow for incremental growth. This means that not only can new servers and new disks be added to the system, but that existing files can be moved to those servers and disks without having to notify the user. The file names that are used by a client should therefore be location independent, in the sense that a file can be moved without changing its name.
- Not only is the capacity of WCS difficult to predict: the uses to which it will be put are similarly hard to define. More precisely, although WCS will assuredly be used for the storage of information, it is hard to say how that information will be accessed. In particular, it is impossible to foresee all of the various index structures that might be necessary. In any case, building an over-general index structure as part of WCS is likely to lead to unnecessary inefficiency. Our solution to this problem is to provide access to WCS files at two levels. A conventional, user-readable hierarchic name system will be supplemented by a flat namespace of machine-generated file identifiers (*fid*): each file version will have a unique fid, and may also have a WCS hierarchic name. If the WCS name structure is inappropriate or inadequate for a particular application, an application-specific index

structure of a more appropriate kind can be built on the space of *fid*s.

The most novel aspect of this design is the notion of file typing, and this is the aspect of WCS that we are exploring in our first prototype. We view the act of writing a file from one machine and later reading it from another as similar to a (very slow) remote procedure call. Like a remote procedure call, a user will be able to write a record consisting of an integer and a character string on one machine, and later read the logically equivalent record onto another. Any necessary byte swapping and character-set translation should be automatic. This can be done, provided that the system is informed which parts of the data represent the integer and which parts the string; this is exactly the rôle of the type definition. The data translation can then be performed by exactly the same mechanism as is used by the RPC system. There is no need to pick a single canonical representation for the data in WCS; multiple, labelled representations can be used, just as multiple labelled representations of data can coexist on the the same communications circuit.

3.5. Mail

We will develop a mail service with two objectives in mind: easing the integration of new hosts into our local environment, and remedying certain flaws in our current mail service. Our development plan will be based on the experiences (and perhaps some of the code) of Xerox's Grapevine system and Berkeley's 4.2 bsd UNIX sendmail utility. We will use the structure of Grapevine's mail servers, the transfer support of sendmail, and our own naming service. RPC-based interfaces will be the mechanism for communicating between mail servers and mail user agents (also known as mail browsers, mail handlers, mail systems, etc.). Our development plan has two phases, the first involving a single mail server and the second involving multiple servers. Among the steps in each phase are: specification of the data (and its format) that we expect to retain in the name server; definition of the RPC interfaces between the user agents and the mail server(s) for both submission and mail delivery; and modification of existing user agents to use these interfaces.

3.6. Printing

The printing service will consist of server programs, which exist on those nodes to which printers are attached and which manage print queues and physical printers, and client programs, which exist on every node that has access to the printing service. A printer database, managed by the name server, provides client programs with location-independent access to printing services. There are three major classes of service: printing service enquiry, print job queuing, and printing service management. Our intention is to model the print system after the current Unix printing system, except that client and server programs will use RPC for the communication of both control information and data, and a comprehensive set of service enquiry functions will be provided.

3.7. Remote Computation

Although it is infeasible to provide general process migration in a heterogeneous environment, we do intend to provide a facility that will enable specific applications to be run remotely. A job that requires, say, substantial computational resources or special hardware, or that can run only on a certain class of machine, can be submitted to the appropriate machine remotely. This remote submission will be as convenient to use as if the program were run locally; in particular, a local interface will take care of collecting the files and local environment information that the remote job needs, shipping this data to the remote machine, and returning the result files. To simplify the construction of such interfaces, we intend to develop a language for describing the input and output requirements of programs; the interface programs will then be generated from descriptions

written in this language. The language will include a system-independent way of describing attributes such as command options. In cases where there are multiple available servers for a needed service, we expect to do simple run-time load sharing.

3.8. Summary

Viewed in the large, our development effort is organised into phases in two different dimensions. First, the higher-level services, such as filing, will be built using the underlying services, such as RPC and authentication. Second, within each service we will progress from minimal to full functionality in a sequence of steps. This means that we will be able to improve our computing environment significantly in a relatively short time, that the services provided in the initial phases will aid implementation of subsequent phases, and that we will benefit quickly from the experience of using our own services.

It is clear that the designs, specifications, and implementations of the underlying facilities on one hand, and of their clients, the four key services, on the other hand, must occur in a roughly alternating sequence. Work on the key services requires some specification of the underlying facilities on which they rely. At the same time, it is not possible to complete the design of the functions and interfaces of the underlying facilities until the needs of their clients (the key services) are more precisely understood. This means that development of the services will cause a refinement in the designs of the underlying facilities, which in turn will affect the designs of the services. This close interplay and the iterative nature of the phased implementation underscore the importance of viewing the project as a whole rather than as separate efforts; our project is truly coordinated in the sense that its components relate closely to one another and cannot be considered in isolation.

4. The Benefits of Accommodating Heterogeneity

Once the services discussed above are in place, we believe that the effort involved in building special-purpose distributed applications will be significantly reduced. We are already seeing the benefits of the heterogeneous RPC mechanism as we begin to prototype mail servers and the file server: it is easier than ever before to build distributed programs that span machine types.

One of our motivating examples was the cost of transforming an isolated machine of a new type into a full participant in our heterogeneous computing environment. Given the above services, we would expect the process to proceed roughly as follows:

- Providing network communication is the first priority. We expect that every system of interest will have Ethernet capability, including a transport protocol such as XNS or TCP. A veneer will be placed on top of the network interface provided by the host operating system, meeting the interface specification required by the RPC facility. The code implementing the RPC runtime support will be ported.

  RPC stub procedures will be produced in one of three ways: by hand, by cross-compilation from another system (the stub compiler produces source-language, not machine-language, programs), or by a stub compiler that is ported from another system.

- Access to the naming service and to the authentication service will involve the production of RPC stubs, but only a very limited amount of application code will need to be ported. This is because our design makes it possible to locate much of the work on server machines, rather than on client systems.

- Once RPC, naming, and authentication are available, the basics for access to the common file system (and, indeed, the other services) will be in place. It will be necessary to port RPC stubs and simple user interface routines for the small number of WCS operations. Although transparent integration of WCS with the existing file system on the new host is

not a goal, the interface routines will access files on this host, so these routines are not directly portable from one system to another.

- Access to the mail service can be provided in several steps. First, a simple user agent is run on mail server machines, and can be accessed using remote login if this is available. Second, a similar user agent will be available for porting, which essentially fronts for the agent on the server machines using a simple RPC interface. Finally, a native user agent can be modified, or one can be ported from another system; a more substantial RPC interface will be required. The mail service itself need not be modified.
- The printing service is also accessed through a simple RPC interface.
- Provision of remote computation requires the existence of a transfer agent that communicates using RPC. Although special-purpose (single application) transfer agents can be built by hand, we will design a generalised interface that can be ported.

5. Conclusion

The problems of heterogeneity, outlined above, arise not just in our own research environment but in almost every state-of-the-art computer research facility. The major exception might be those laboratories whose equipment is limited to that which is available from a single vendor, or that which will run a single operating system. However, such limitations should not be imposed simply for convenience in managing the facility: they imply either corresponding limits on the research that will be performed, or greatly decreased productivity due to the absence of high-level testbeds, or both. We therefore believe that the accommodation of heterogeneous computer systems – coupling dissimilar systems into an existing computing environment – is a critical problem facing experimental computer research establishments.

We are working on broad solutions to the various problems that arise in accommodating heterogeneity: naming, authentication, mail, printing, and remote computation, in addition to basic interconnect (RPC) and filing. The synergism arising from this larger context – the *Heterogeneous Computer Systems (HCS)* project – is a major strength of our effort.

Bibliography

[1] Almes, G. T., Black, A. P., Lazowska, E. D. and Noe, J. D. The Eden System: A Technical Review. *IEEE Trans. on Software Eng. SE-11, Nr 1* (January 1985), pp43-59.

[2] Black, A. P., Lazowska, E. D., Levy, H. M., Notkin, D., Sanislo, J. and Zahorjan, J. An Approach to Accommodating Heterogeneity. Tech. Rep. 85-10-04, University of Washington, Computer Science Dept, October 1985.

[3] Ching, D. T. A Remote Procedure Call Facility for Accommodating Heterogeneity. M.S. Thesis, University of Washington, Computer Science Dept, August 1986.

[4] Gettys, J. Project Athena. *Usenix Association Summer Conf. Proc.*, Salt Lake City, June 1984, pp72-77.

[5] Kapauan, A. A., Field, J. T., Gannon, D. B. and Snyder, L. The Pringle Parallel Computer.. *Proc. 11th International Symp. on Computer Architecture*, June 1984.

[6] Oppen, D. C. and Dalal, Y. K. The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment. *ACM Transactions on Office Information Systems 1, Nr 3* (July 1983), pp230-253.

[7] Satyanarayanan, M. The ITC Project: An Experiment in Large-Scale Distributed Personal Computing. CMU-ITC-035, Information Technology Center, Carnegie-Mellon University, Pittsburgh, PA, October 1984.

[8] Sun Microsystems. *Remote Procedure Call Protocol Specification.* Sun Microsystems, Inc., January 1985.

[9] Xerox Corporation. Courier: The Remote Procedure Call Protocol.. Tech. Rep. XSIS 038112 , December 1981.