HETEROGENEOUS COMPUTING ENVIRONMENTS: REPORT ON THE ACM SIGOPS WORKSHOP ON ACCOMMODATING HETEROGENEITY*

The ACM SIGOPS Workshop on Accommodating Heterogeneity was conducted in December 1985 in Eastbound, Wash., as a forum for an international group of fifty researchers to discuss the technical issues surrounding heterogeneous computing environments.

DAVID NOTKIN, NORMAN HUTCHINSON, JAN SANISLO, and MICHAEL SCHWARTZ

INTRODUCTION

A heterogeneous computing environment consists of interconnected sets of dissimilar hardware or software systems. Because of the diversity, interconnecting systems is far more difficult in heterogeneous environments than in homogeneous environments, where each system is based on the same, or closely related, hardware and software. Examples of heterogeneous environments include: a network with 3 VAXes, 16 SUNs, and 1 Symbolics LISP machine; a network with 1 DEC-2060, 1 IBM-4341, and 20 IBM PC-ATs; a network with 12 Xerox D-Machines, 6 of which are running Interlisp and 6 of which are running XDE. In contrast, examples of homogeneous environments include: a network of Macintoshes linked together with AppleTalk; a network of MicroVAXes running Ultrix; a network of SUNs running UNIX and NFS; a network running Eden [1]; a network running Locus [12].

Heterogeneity is often unavoidable. It occurs as evolving needs and resources lead to the acquisition or development of diverse hardware and software. As a computing environment evolves, there is a tension between retaining homogeneity and acquiring new types of systems. Since some efforts are best conducted on systems different from those already available, this tension must at times be resolved in favor of heterogeneity. For example, research on constraint-based animation may be easier to perform on a Smalltalk engine than on a more conventional workstation environment.

Heterogeneity can be approached in many ways; each style arises from a specific set of underlying assumptions. Examples: Should a particular system characteristic, such as distribution, be hidden? Should a low-level facility, such as remote procedure call (RPC), be provided in all systems? Is a particular feature, such as transparent network file access, worth the added development cost? How much heterogeneity does the style anticipate? Different assumptions appropriate for each style of heterogeneity lead to different technical issues and problems.

Problems due to heterogeneity arise in several specific areas:

* A preliminary version of this report appeared in *Operating Systems Review* 20, 2 (Apr. 1986), 9–24, and also as Technical Report 86-02-01, Department of Computer Science, University of Washington (Feb. 1986). The report printed here is not a transcript: the order of the discussions has been changed, remarks have been paraphrased, and contents have been condensed. However, an attempt to remain faithful to the proceedings has been made.

Support for preparation of this report was provided in part by NSF grant DCR-8420945.

Interconnection. How should heterogeneous systems communicate? Is message passing or remote procedure call the more suitable communication paradigm? How can systems and languages with different data representations (such as byte-ordering or record layouts) be accommodated?

Filing. What kind of file system is needed in a heterogeneous environment? Should the file system support typing? When heterogeneous systems share data through a file system, where are the required translations done?

Authentication. How is authentication supported in a heterogeneous environment? What are the sources of distrust and diversity in such an environment? How is local system autonomy over authentication provided in heterogeneous environments?

Naming. How is naming provided in heterogeneous systems? What objects can be named across systems? How are they named? How does the environment evolve as new systems and naming approaches are incorporated?

User Interfaces. How are varied user interfaces accommodated and shared between heterogeneous systems? Do you port an application? Do you provide a veneer so that it appears that an application is running on another machine? Do you split the user interface from the basic application and run these on separate systems?

STYLES OF HETEROGENEITY

There is not a single, correct way to address the problems of heterogeneity. Instead, there are many possible different styles, each driven by its own set of underlying assumptions and objectives. During the Workshop we identified basic assumptions and approaches participants use in their work. Without question, there are other styles of heterogeneity that were not represented by participants at the Workshop and are therefore not presented here.

Loose Integration Through Network Services

Accommodating heterogenous computer systems in this style is motivated by an environment of a large number of system types and a small number of instances of some of these types. For example, such an environment might have VAXes and SUNs running UNIX, one or two Symbolics LISP machines, and a number of prototypes of special purpose architectures. Here, the current cost of accommodating new systems is great.

A group at the University of Washington is investigating accommodating this style of heterogeneity [4]. Their approach to these problems is to reduce the cost of introducing a system and allowing it to use basic facilities (RPC, naming, and authentication) and services (filing, mail, printing, and remote computation). In general, transparent use of these facilities and services is not necessary in this approach, although it would be possible in cases where both economics and source code availability permit. Instead, the approach is to construct an environment based on simple clients and sophisticated servers. It should be inexpensive to develop a new client to take advantage of existing servers.

Sharing Among Different Languages Cultures

A second style of accommodating heterogeneity is based on a desire to share programs written in radically different programming languages, to increase the reuse of programs among groups of research programmers with different computing cultures, such as LISP and CLU. In particular, one culture's programs should be able to invoke another culture's programs in a transparent manner. This style expects a large number of instances of each system type. Hence, the effort spent on accommodating each system type can be greater than in the loose style of integration previously described.

A group at the Laboratory for Computer Science (LCS) at MIT is studying this style of accommodation. Their approach relies on two components: an invocation mechanism and a set of interfaces defining shared services. For invocation, the LCS group is considering an RPC facility that supports caller-initiated aborts, procedure parameters and callback, exception handling, failure semantics, atomicity, abstract types as parameters, a definition language for types and program interfaces, and authentication. For the second component, the LCS group plans to include name servers, object stores, archival stores, an authentication server, and a facility for cataloging programs, interface stubs, and abstract data types. This catalog contains converters and checkers in addition to object definitions.

Front-Ends for Multiple, Existing Systems

Another style of accommodating heterogeneity considers an environment in which there are multiple, existing systems over which there is no control and that cannot be changed, for example, using PCs to access an existing corporate database. By adding an understanding of the database to the PCs (which can be changed), the systems will be able to accommodate the database in the PC environment. A "protocol generator" for user interfaces might help in this style. Dave Reed of Lotus Development Corporation introduced this style at the Workshop.

Transparent Operating System Bridges

This style arises in an environment of several different types of workstations sharing resources via a common set of network backbone machines (e.g., a collection of PC-DOS machines, Macintoshes, and UNIX workstations served by a backbone of UNIX or Locus machines [12]). The capability of each type of workstation is extended by transparent access to remote resources, but the remote resources appear to be those of each particular workstation, rather than necessitating users of a particular type of workstation to understand the properties of the backbone machines. At the same time, the workstation user should be able to take advantage of the backbone machine's unique capabilities whenever desired.

The Distributed Systems Laboratory at UCLA is pursuing accommodating heterogeneity through transparent operating system bridges (TOSB). These objectives are achieved by intercepting operating system calls on the local system and passing appropriate calls to a server process on the remote system for fulfillment. Transparent access to remote resources implies that programs designed for a particular workstation environment can take advantage of remote resources without program modification. The most important case is transparent access to a remote file system; however, transparent operating system bridges can support a spectrum of services including local programs directly accessing remote files, local programs invoking remote processes, communication between local and remote processes, and remote processes directly accessing local files. There are some general principles for constructing a TOSB, but each pair of operating systems provides unique challenges, and solutions for them do not tend to be very general. Thus, the TOSB approach is best suited to environments with relatively few different types of operating systems.

Coherence

Coherence carefully defines a layer of software so as to enforce uniformity and permit implementation on diverse hardware to accommodate heterogeneity. Because the costs of providing coherence are great, coherence is feasible only in environments with a small number of system types with a large number of instances of each type. This style has been adopted in some instructional environments.

CMU's ITC project [8] and MIT's Project Athena [2, 6] exhibit coherence most clearly. In the ITC project, coherence is primarily at the level of the logically centralized file service. In Project Athena, coherence is primarily at the applications programming interface. Both projects rely on a uniform underlying operating system, UNIX, and on their window systems, each local products.

BASIC TOPICS IN HETEROGENEITY

The bulk of the sessions focused on specific areas that must be considered when dealing with heterogeneity. Distilled discussions on these topics—interconnection, filing, authentication, naming, and user interfaces—follow.

Interconnection

ISO transport was too low a focus. The discussion of interconnection of heterogeneous systems gravitated to a discussion of the proper way for processes running on different nodes to communicate. The two basic mechanisms for program communication, *message passing* and *remote procedure call* (RPC), were discussed. Message passing consists of passing a message asynchronously from one process to another, such that both the sending and receiving processes proceed concurrently. RPC, as defined by Birrell and Nelson [3, 10], provides semantics across a network that are nearly identical to those of procedure call in a standard programming language: the RPC is synchronous, the caller blocks until a reply is received or the call is aborted. In message passing, the data usually appear to the system as a stream of bytes, while in RPC the data have some structure and are type-checked. The sending or calling process is generally called the *client*; the receiving or called process is generally called the *server*.

Figure 1 illustrates how RPC usually works. The client is written as if it called the server directly using conventional procedure call mechanisms. To simulate this relationship across the network boundary (represented by the striped line down the center), two *stubs* are needed. The client stub's interface is identical to that of the server; the server stub's interface is identical to the client's. The client stub is responsible for translating the arguments into a suitable format for transmission over the network and also for passing the converted arguments to the transport mechanism. The server stub, conversely, is responsible for receiving the arguments from the transport mechanism and converting the arguments into the server's format. Multiple calls may take place between the stubs and the transport mechanisms, depending on the actual RPC implementation. Just as procedures must be linked before they can call one another, it is necessary to *bind* clients and servers together before RPC can take place.

Although the synchronous nature of RPC is suitable for many applications, a mechanism is needed to permit concurrent execution. *Light-weight processes* (LWPs), which permit a single program to define multiple threads of control, are the conventional solution. LWPs share a single address space, allowing context swaps between LWPs to be done much more quickly than traditional process swapping. Combin-

ing RPC and LWPs is natural: each remote call is embedded in its own LWP and when that call blocks, another LWP is scheduled. Hence, threads of control in both the calling and called process are active.

One point of view was that the RPC paradigm provides an appropriate level of abstraction for communicating between programs across nodes. Given this strict definition of RPC, the question became: "Are these semantics sufficient?" The answer was, in general, yes. However, there are times when a more flexible model of communication is mandatory. Examples of such instances are asynchronous operation when LWPs are not available, and a "no-reply" option when the reply would contain essentially no information. For instance, when a display is updated, the program sending the data need not wait for a reply from the output unit. There was some discussion as to whether pure semantics could be maintained given that light-weight processes were available. The answer was a qualified yes; however, performance will probably be reduced in "no-reply" situations.

The discussion of RPC as an acceptable communication paradigm then shifted to problems directly associated with heterogeneity. Several areas that require flexibility due to heterogeneity were identified. Transport protocols, such as TCP and XNS, differ across networks; how is it decided which protocol client and server processes will speak? Data representations, such as byte-ordering and the layout of structured data, differ from machine to machine and compiler to compiler; how are the necessary transformations identified and applied? Semantics and type systems vary from language to language; how can RPC semantics be maintained between languages that are dissimilar in this regard?

First consider data representation. There are at least three ways to select a data representation for transport. First, define and use a single standard representation. Second, send the data and require the receiving side to understand it. Third, negotiate a representation at bind time (i.e., when a specific client and server first decide to communicate). For transport protocols, variations of the first and third options are possible (the second is not possible for transport since a common transport is required to support the initiating conversation).

The problem with the single-standard approach is the potential for unnecessary inefficiency. The most obvious example is two systems with the same byte-ordering would be required to communicate by swapping and then unswapping bytes if their mutual byte-ordering differed from the standard. Experience with the DEC/SRC RPC system demonstrated the potential for selecting transport protocols and data

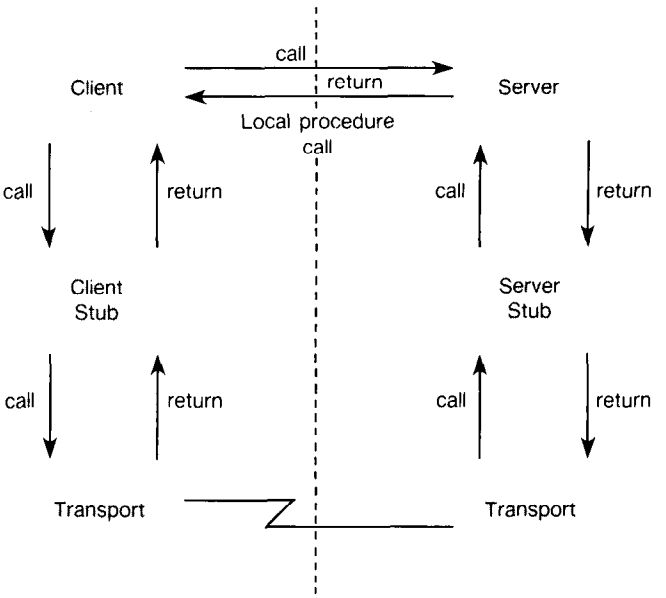

FIGURE 1. Remote Procedure Call

representations (within a limited domain) at bind time.

The relationship between efficiency and uniformity, with respect to data representation, was discussed at length. On one hand, some people were willing to accept a degree of ineffciency in the area of data representation as the price of simplicity. Bill Joy of SUN contended that the work required to minimize byte-swapping, for instance, is of little benefit since only as little as 5 percent of run-time is spent filling in data packets. On the other hand, in many cases it may be possible to reduce these costs quite easily.

The heterogeneity imposed by varying semantic and type models is usually addressed through the use of stubs to make RPC look like conventional procedure call, as shown in Figure 1. Stubs can be quite complex as they are usually responsible for packaging data and communicating with the transport layer. To relieve the user from writing complex stubs, stub generators are often provided with RPC systems. Although stub generation usually accounts for syntactic differences among languages, making the semantics compatible with all existing languages is at best difficult, and making them compatible with yet-to-be-developed languages is impossible.

Interface description languages (IDLs) are often a basis for generating stubs [7, 13]. Is it possible to define and use a single IDL? The adoption of a single IDL does not preclude hand-coding of stubs for particular applications or esoteric requirements. The

possibility of several classes of IDLs for separate classes of languages was raised.

Filing

Andrew Black of the University of Washington introduced filing with a chart (Figure 2) that categorized some existing distributed file systems. (A survey on distributed file systems appears in [11].) He showed that the design space has many dimensions and that file system designers made different decisions in each dimension. Several of these dimensions with respect to the effects of heterogeneity were considered.

A large part of the discussion focused on file properties, particularly typing. On one hand, at a certain level, files are all the same "type"—simply collections of bytes or blocks. On the other hand, all files are implicitly typed in the sense that programs that access a file make assumptions about the nature of the data. If these assumptions are wrong, the data may be misinterpreted: this is a type error. If files are typed, then such an error can be detected before it leads to a rubble of bits.

The UNIX file system is an interesting study of file typing. The UNIX abstraction of an uninterpreted sequence of bytes is a great simplification; programmers must provide any further abstractions at a higher level. This abstraction makes some tasks easier but others harder. For instance, UNIX records must be constructed and shared by unenforced convention. On the other hand, generic utilities are easily written since there is only one file type.

Typing is more of an issue in a heterogeneous environment because different machines use different data formats, for example, different character codings and byte orders. Another reason is simply a larger number of file structures. If a file is typed, the file system can do the appropriate data conversion; if not, the client must do its own conversion. Another option is to provide self-describing data types, that is, objects that carry their type information with them. The advantage is that only the applications that deal with a specific type need to know about the type.

The relationship between file typing and the data representation problems of RPC were discussed. Files can be viewed as providing "time-shifted" communication, a little like RPC over a delay line. Because the reader and writer do not communicate with each other directly, the file system should have the responsibility of communicating the information an RPC system would exchange at bind time, and of typing the data in the same way an RPC system might. This can be achieved either by translating the file contents into a common intermediate format, or by recording the data in the sender's format and recording explicit formatting information.

The degree to which files are shared affects design decisions in a file system. To make these decisions properly, it must be determined whether the sharing supported by a file system is actual or just potential. In Multics, there is virtually no short-term sharing. Measuring sharing patterns before making decisions was suggested. Such measurements may be deceptive, however, since the infrequency of actual sharing does not imply a lack of need of actual sharing.

Several other questions were raised and briefly addressed. Files are usually addressed by name; how can heterogeneous file systems conform with diverse naming systems? What happens when applications demand more from a remote file system (e.g., locking and record access) than the remote system can provide? Is the notion of "file" too restrictive for the diverse environment we anticipate? (Although an object-oriented approach was suggested as more profitable, the fact remains that existing file systems are not, for the most part, object-oriented.)

Dave Reed of Lotus pointed out an anomaly. There is great diversity in file systems, but the Andrew system [8], in an approach shared by many other efforts, uses a single file system of its own design as the "glue" that connects heterogeneous components. This scheme relies on *replacing* the existing file systems with the new "glue" file system. But what is to be done at the next level up, when the Andrew file system needs to be connected to other similar systems? Presumably, at this level, we are not prepared to discard the file systems and build a new system that acts as "superglue." We may therefore be forced to provide remote access to a number of existing file systems rather than a single common file service where a file must live if it is to be shared.

Authentication

Discussions of authentication and authorization in heterogeneous computer systems focused on classes of problems rather than on specific authentication mechanisms. Three broad problem areas were covered: (1) sources of distrust and diversity with respect to authentication; (2) identifying the actual function of authentication and authorization systems; (3) accommodating the need for local system autonomy within global authentication environments.

Sources of distrust in heterogeneous systems include networks, gateways, hardware, operating systems, run-time systems, application programs, students, fellow researchers, family, and yourself. Sources of diversity include hardware (especially encryption support), programming environments, the

	Read Only?	Universal/FSF?	Transparent Access	Location Independent Names	Replication	Caching	Fetch Grain
Sesame	yes	Universal w/i Spice FSF for world	yes w/i Spice no for world	file id	optional	whole file	pages
IBIS	no, Unix locks	Universal	yes (library)	no, planned	on demand	yes	pages
Tilde	no, Unix locks	Universal	yes	at tree level	no	yes	pages
Xerox IFS	no (transactions added)	FSF	n/a	no	no (added)	n/a	pages and streams
Cedar	yes	FSF	yes	no	no	whole file	whole file
Sun NFS	no, no locks	FSF, but every Sun w/s can be FS	yes	no	no	no	pages
Vice/Virtue	no, locks	FSF	yes	file id	limited	yes, invalidate on read (now write)	file
Juniper (XDFS)	no, transactions	FSF	n/a	no	no		
Alpine	transactions	FSF	n/a	yes	n/a	no	n/a
Apollo	no, timestamp version consistency	Universal	yes	file id	no	V.M. Cache	pages
Glasser/Ungar	yes (really)	Universal	yes	no	no	no	pages
Eden	no, transactions	Universal w/i Eden	access from Eden only	file capa	yes	no	invocation
Amoeba	no, optimistic CC and locks	FSF	yes (through rose colored glasses)	file capa	yes	pages	var size pages
Roe	no, locks	FSF, stored on clients	no	yes	yes	migration	stream
8th Edn NFS	no, no locks	Universal w/i Unix	yes	no	no	no	stream
Locus	no, Unix locks	Universal	yes	yes	optional	pages	pages

The columns have the following meanings:
Read Only? Are files read only (yes) or overwritable (no). If no, what mechanism is used to prevent conflicting writes?
Universal/FSF? Does the file service provide universal access to files in existing file systems, or does it provide a new kind of file (a File Server File, FSF) that must be created explicitly?
Transparent Access. Do the host operating systems hide the difference between accessing the local and distributed file systems?
Location Independent Names. At what level (if at all) is the name of a file independent of its location?
Replication. Is replication a standard feature, an option, or unavailable?
Caching. Is caching performed? If so, what is the unit of caching?
Fetch Grain. How is the file fetched from the server?

FIGURE 2. Comparison of File Systems [4]

class of problems being solved, tolerance of costs, protocols for supplying and using authentication information, sheer size, and different administrations.

Although there was agreement on where problems originate, there was hot debate on whether the goal of an authentication mechanism was punishment or prevention. If punishment were the ultimate goal, then relatively passive mechanisms in conjunction with logging and auditing could be used to record information permitting the identification and apprehension of offenders. Prevention requires that more complex, active mechanisms be used to control

execution of undesirable actions, malicious or inadvertent.

The punishment approach was criticized on the basis of the difficulties that arise in trying to track down operations spanning more than one "boundary." The moral and administrative implications of forming a "network police force" to implement punishment were also considered serious problems. It was suggested that the real world functions by auditing and logging, and that computer systems will have to fit to human systems, and not vice-versa. The problem of authenticating auditing information was mentioned in this regard.

As an example of a middle ground, Jerry Saltzer of MIT stated: "Project Athena is building an authentication server primarily because each private workstation is owned by a student, and each public workstation is captured by indivdiual students as superusers. Given this situation, there must be a way to protect the services, such as mail, printers, and file systems, from inadvertent errors. The goal is to halt mistakes but not necessarily malice."

The prevention approach requires pairwise agreement between each two communicating entities. Roger Needham of the University of Cambridge pointed out that this approach is cumbersome; he and Michael Schroeder showed how to optimize it in a homogeneous environment through the construction of a global authentication service with a distributed implementation [9]. Some problems, such as making sure to avoid using untrustworthy authentication services, become far more serious in a heterogeneous environment.

Other problems arise because different environments often have different views of the level of protection that is necessary or desirable. Further, different authentication or authorization boundaries may exist within a single system (e.g., within a university laboratory different rights might be provided depending on whether the user was accessing a research or an educational subnet). Deborah Estrin of USC observed that any authentication scheme for heterogeneous environments will require cooperation between autonomous administrative units. In this respect, there are very strong parallels between the problem of authentication and the problem of naming.

Authentication and authorization mechanisms are usually intimately related to the local operating system, relying on being "built-in" to both prevent and detect tampering. Is it possible to accommodate such low-level OS dependencies in a distributed, heterogeneous environment?

Rick Rashid of CMU enumerated classes of solutions to the authentication problem: building appropriate size barriers to discourage casual breaches; logging activities at each node; performing cross checks at intervals to ensure consistency; instituting a "network police force"; educating and applying social pressures to users; and punishing those found guilty.

Rashid also presented a short discussion of authorization. The key point was that providing a solution to the authentication problem is only half the battle. The authentication information must then be interpreted in a consistent manner across systems. A mechanism for performing this interpretation is a separate problem that is at least as hard as the original authentication.

Naming

John Zahorjan of the University of Washington identified four issues to be considered in naming in heterogeneous systems: accommodation of evolutionary growth, name resolution, transparency, and name acquisition. The discussion illustrated an underlying theme of the workshop: We know how to provide many styles of services, but which are the "right" ones? And, can multiple "right" approaches be combined smoothly?

In naming, several separate dimensions are apparent. One key issue is whether names should be *relative* or *absolute*. An absolute name refers to the same object regardless of the "context" (that is, the site, the user, and possibly the application) in which it is issued. Absolute names facilitate sharing since they provide a common vocabulary with which to refer to objects. A relative name is context dependent. A common example that illustrates the utility of relative names is mail nicknames. Each user creates a set of easily remembered nicknames to be used in place of more cumbersome, network-dependent mailbox names. The nickname `leach`, for example, is much easier to remember than the complete name of `apollo!pjl@uw-beaver.edu`. Another example of relative names is file names in a shared file system. A standard mechanism for providing these names imports or mounts a portion of a foreign name space and attaches it to a local "root." (E.g., this model has been used in the Andrew file system [8].) A major advantage of relative names is that convenient names for objects can be chosen within each context independently of other contexts. Particularly in a heterogeneous environment, this flexibility is a great asset since different contexts may have fundamentally different requirements of the naming scheme.

A notion closely linked to that of absolute and relative names is whether there is a single global, homogeneous name space or many local name

spaces. A global name space appears to be desirable, but the cooperation and extent of changes required to implement it are considerable, especially in a heterogeneous environment. Indeed, the environment may be heterogeneous in part because individual subsystems in the environment might prefer or require their own naming schemes. It was observed, however, that if there is no global name space, then it is not possible to name all objects in all name spaces because some naming environments will have no way to translate some names.

Because sharing is so important, most existing name services provide absolute names. However, distributed, heterogeneous environments (among others) usually provide for some style of relative names as well. For example, in Locus a user may invoke a computation without knowing on which machine it will run. Thus, a mechanism is required whereby a single name can refer to any one of a number of executable files, each one appropriate to a different system type. Similarly, some names benefit significantly from being relative, such as the use of /tmp to refer to a temporary directory in a distributed UNIX system.

A number of short presentations on aspects of naming were given. David Cheriton of Stanford University described naming in the V system [5]. In contrast to most systems, which present a single logically centralized service, name management in V is distributed among the objects responsible for the named entities. This can be an advantage, especially in heterogeneous systems where name syntax and operations may differ significantly from one site to another.

Thomas Murtagh of Purdue University introduced his notion of *nice* names, that is, names that are location-transparent and symbolic, that can be used as syntactic sugar to insulate the user from the "nastiness" of the actual underlying naming scheme. Murtagh said that nice names are local, not global, and are required by the needs of application programs. Dave Clark of MIT suggested that nice names might work in a distributed, universal name service, assuming that they can be transported appropriately (that is, that they can really be kept location-independent). The possibility of using nice names is generally a function of facilities available in the command language, rather than the operating system primitives. Even then, nice names can suffer from the drawbacks of relativism, that is, path compression, and finding alternate paths may be difficult to do with nice names.

Karen Sollins of MIT discussed administrative issues that arise in handling name services. In a typical hierarchical name space human "managers" are responsible for subtrees of the name space at various levels in the naming tree. Because the name service provides some of the keys required to access the named resources (and in some cases all of the keys), it may be necessary to give control of access and update authority to the local manager. This makes managing the overall name service more difficult, as it may not be possible to make changes uniformly to all supporting servers.

Clark observed that so far little attention has been given to the dynamic aspect of naming. The autonomy that is characteristic of heterogeneous systems requires that there be provision for recovery from system failures and on-line changes to the name space (e.g., changed mailbox route or reincarnated object). In some sense, the discussion focused more on a name-management system than on a name space.

User Interfaces

Mark Weiser of the University of Maryland introduced this topic by observing that user interfaces are qualitatively different from the other basic topics: every system has one, they cannot, by definition, be hidden from users, and they are impossible to construct as a central service.

There are three ways to accommodate user interfaces. The first is porting, where an application is moved to the system being accommodated. The second is masking, where the application appears to have been ported, when in fact it is actually running entirely on another machine. The third is mapping, where the user interface is moved, but the heart of the application is not; the characteristics of each system must be mapped to the other.

Weiser defined four levels of user interface heterogeneity: (1) what the user sees, (2) what the application program sees and provides, (3) what the window system sees and provides, and (4) what the hardware provides. Different means of accommodation are more applicable at these different levels. For example, porting is a natural means of accommodating the interface between levels three and four, but mapping is more appropriate between levels two and three.

There was contention over the future of window systems in a heterogeneous environment. One side argued that they are too big and it is too difficult to integrate applications into a window system, stating that sophisticated applications almost always want to use the screen in a "raw" mode. This side continued by stating that every window system that goes into operation terminates work on user interfaces for at least one and a half years because most systems make too many decisions about the user interface,

which locks out innovation. The other side responded that there are examples of sophisticated applications (such as Interleaf's WPS product on Apollos and SUNs) where raw mode was not needed to get the efficiency necessary for success.

Any window system that accommodates heterogeneity must be able to support different user interfaces (e.g., tiled and overlayed), different program interfaces (e.g., X and Sun Windows), and new input paradigms (e.g., natural language, speech, and images). The possibility of supporting all this by adopting a standard, extensible protocol is being explored.

Another approach, described by Keith Lantz of Stanford University, promotes the workstation as a front-end to all available resources, both local and remote. This way, the user is insulated from the underlying heterogeneous system. The interaction with all resources, since it is handled by local software, is consistent and natural. The user interface must support four levels of interaction: terminal management, command interaction and response handling, application specific interaction, and multi-application interaction. Additionally, the user must be permitted to configure the software components of the system to meet individual preferences.

Acknowledgments. Ed Lazowska and John Zahorjan put much time and effort into helping create and revise this report for *Communications*. The other members of the Heterogeneous Computer Systems project at the University of Washington, including Andrew Black, Dennis Ching, Henry Levy, John Maloney, and Mark Squillante contributed as well. Dave Clark, Terry Gray, Paul Leach, and Mark Weiser supplied comments, suggestions, and (in some cases) text for the report. Peter Denning encouraged us to submit the report to *Communications* and then made several useful suggestions on how to improve the report. Thanks also to Andrew Birrell, Bill Joy, Jim Morris, and other members of the organizing committee. Of course, thanks go to the participants of the workshop.

REFERENCES

1. 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).
2. Balkovich, E., Lerman, S., and Parmelee, R.P. Computing in higher education: The Athena experience. *Commun. ACM 28*, 11(Nov. 1985), 1214–1224.
3. Birrell, A.D., and Nelson, B.J. Implementing remote procedure calls. *ACM Trans. Comput. Syst. 2*, 1(Feb. 1984).
4. 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, (Oct. 1985).
5. Cheriton, D.R., and Mann, T.P. Uniform access to distributed name interpretation. In *Proceedings of the 4th International Conference on Distributed Systems*. (May 1984).
6. Gettys, J. Project Athena. In *USENIX Summer Conference Proceedings*. (June 1984).
7. 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 Languages*. (Jan. 1985).
8. 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.
9. Needham, R.M., and Schroeder, M.D. Using encryption for authentication in large networks of computers. *Commun. ACM 21*, 12(Dec. 1978), 993–999.
10. Nelson, B.J. Remote procedure call. Ph.D. dissertation, Tech. Rep. CMU-CS-81-119, Dept. of Computer Science, Carnegie-Mellon Univ., (May 1981).
11. Svobodova, L. File servers for network-based distributed systems. *ACM Comput. Surv. 16*, 4(Dec. 1984).
12. Walker, B. et al. The LOCUS distributed operating system. In *Proceedings of the 9th ACM Symposium on Operating System Principles*. (Oct. 1983).
13. Xerox Corporation. Courier: The remote procedure call protocol. XSIS 038112, (Dec. 1981).

CR Categories and Subject Descriptors: C.2 [**Computer Systems Organization**]: Computer Communication Networks; D.4 [**Software**]: Operating Systems

Additional Key Words and Phrases: distributed processing, heterogeneity

Contact: David Notkin, Department of Computer Science, FR-35, University of Washington, Seattle, WA 98195.