Edgar H. Sibley
Panel Editor

Heterogeneity of hardware and software is a fact in most distributed computing environments. The DACNOS prototype is a network operating system that enables resource sharing in such environments. It extends the local operating systems without interfering with existing programs. It provides comprehensive system level support for distributed applications.

Retrospective on DACNOS

Kurt Geihs and Ulf Hollberg

In DACNOS (Distributed Academic Computing Network Operating System) we have addressed two characteristic aspects of today's computing environments: distribution and heterogeneity. While the former aspect is willingly accepted as a move to more power and flexibility for the user, the latter is in many environments a—sometimes unwanted, mostly functionally required—fact of life. Many people have studied distributed systems for the special case of a *homogeneous* environment. DACNOS has aimed at providing efficient and convenient support for the cooperation of *heterogeneous* computing systems.

Heterogeneity in systems is primarily because there is no single hardware and software architecture that serves all computing purposes equally well. Heterogeneity is apparent in different machine hardware architectures, operating systems, networking facilities, and user access control, to name those who were considered for the DACNOS design. Other types of heterogeneity are conceivable, (e.g., user interfaces, application subsystems, or even multiple information media). These were not considered for the current prototype; however, they seem to make interesting areas for further research.

Our initial target environment was the computing infrastructure of the Computer Science department of the University of Karlsruhe in West Germany.[1] There we found, among others, IBM/370 computers running VM/CMS, DEC/VAX computers under VMS,[2] and IBM PCs running PC-DOS. These operating systems and machine architectures differ greatly in their hardware, software, and interface concepts. There was file transfer between the host computers, and some PCs were linked to the IBM hosts supporting terminal emulation, but there was no *resource sharing* between applications running on different machines. Data generated on one machine had to be shipped in a separate step to another machine in order to process it there in an application—not to mention services such as directories or transparent file access across the distributed computers. (Even today, this situation still is very typical for many data processing environments of large organizations.)

The DACNOS prototype was first implemented on VM/CMS, VMS, and PC DOS. This prototype has been studied extensively and used to build various distributed applications. It has also been ported to two more operating systems, i.e., AIX on IBM PC RT and IBM PS/2 and OS/2 on IBM PS/2. In this article we discuss the fundamental design assumptions of DACNOS that reflect our systematic approach to solve the heterogeneity problem as well as our experiences with implementing a prototype on top of five operating systems. For obvious reasons we cannot elaborate on all aspects and components of DACNOS in appropriate detail. The cited references should provide more information on a particular DACNOS subject. The section on "Goals and Implications" describes our design goals and constraints. The section on "Architecture" shows how the DACNOS architecture reflects these goals. The section on "Application Experiences" contains some examples for real-life applications that were built on top of DACNOS. In the section on "Implementation" we discuss what it takes to implement DACNOS on a system and how easy or hard the portation was for the above-mentioned operating systems. We also provide performance data for some scenarios. In the section on "Discussion of Related Work" we compare DACNOS to related work on heterogeneous distributed systems. The last section contains our main conclusions and looks toward future extensions.

GOALS AND IMPLICATIONS

Resource sharing between heterogeneous autonomous computers has been the focus of our research. Fast networking hardware and low-level communication software was available to the academic community on the

[1] The DACNOS effort was part of a cooperation project of the University of Karlsruhe and IBM Germany. The project's goal was to study the application of computers for the support of academic teaching and research [12].
[2] DEC, VAX and VMS are registered trademarks of Digital Equipment Corporation.

campus, but it was still a very cumbersome and often replicated task to write an application that integrated services from several computing devices, although the need for such applications seemed to increase steadily. Our intention was to provide the programmer of a distributed application with convenient system support to facilitate the controlled access to distributed computing resources—basically the same support he or she is used to when writing a local application. This *local-system paradigm* was our guideline for many design decisions.

Two as-much-as-possible goals stood at the beginning of our design considerations:

(1) the application programmer should not have to deal with low-level details of operating systems and communication protocols and
(2) heterogeneity should be handled by the network operating system and be hidden from the application programmer.

In other words, we wanted to provide a high degree of insulation from both distribution and heterogeneity. These two goals led to the design of an *application platform* for programmers of distributed applications. This platform is called "*Remote Service Call* (RSC)." It is the key component in the NOS kernel that provides an application-oriented (as opposed to a communication-oriented) interface to the cooperation of heterogeneous computers. How the "application orientation" is reflected in the RSC interface will be shown in the next section.

Since the interconnected computing systems are part of many rather independent organizational university structures having individual application requirements and solutions, (e.g., research institutes, library services, and student workstation pools), the following two design constraints were significant for the NOS design:

- do not replace the individual operating systems and
- retain the autonomy of the involved systems.

We could not and did not want to (and many commercial users would agree) enforce a single operating system, e.g., UNIX,[3] on all machines. (Several UNIX-based distributed systems can handle heterogeneous hardware architectures [3, 17, 20].) Such a step would have made a large base of applications and investments worthless. Consequently, our NOS is an add-on to the different operating systems that does not interfere with existing applications, but makes it feasible to have access to remote resources in formerly only-local applications. In many cases this remote access is transparent to the application software. (See the section on "Application Experiences.")

When cooperating with remote partners, a DACNOS node does not give up its right to decide autonomously about the access and the management of its resources. For example, access to a resource has to be granted explicitly; it can be revoked at any time; and the allocation of resources to requestors is completely up to the provider of the service. This property distinguishes the DAC Network Operating System from many distributed operating systems where the nodes relinquish some of their control autonomy to become part of a global "whole." The emphasis on autonomy does not preclude a DACNOS-wide management support that helps to allocate and control available resources in a desired manner [7]. It requires mechanisms for access control and protection across distributed computers.

When discussing software add-ons to heterogeneous systems: portability must be a design goal—not just for applications but also for the NOS software. Besides being good software engineering practice, portability is essential for a system that is to be ported to many heterogeneous computers. In our opinion we could otherwise not claim to have a *systematic* approach to overcome heterogeneity. To achieve portability one has to define a software module structure that clearly isolates system dependent and independent components. In so doing, the portation effort is reduced to the modification and adaptation of only a few components. A good modular structure will obviously also support the manageability and extensibility of such a rather large software complex.

ARCHITECTURE

Cooperation in DACNOS is conceived as client-server interactions. Clients and servers reside in *logical nodes*, which are mapped onto the (physical) nodes of the underlying communication network. A logical node has a network transport address and is typically associated with an operating system process or process group having a single address space. In VM/CMS each virtual machine would be a logical node, while a single-user PC is considered a single logical node. In general, a logical node corresponds to a user process on a computer. It is the smallest addressable unit in the transport system.

Figure 1 shows the structure of a system with a DACNOS extension. Applications make use of the host operating system services as before and can now access remote resources through calls to the NOS System Services. Applications may also directly call operations offered by the NOS kernel's interface, i.e., the Remote Service Call interface. This is the lowest NOS access level. The components drawn below RSC in Figure 1 are not visible to the "outside."

Remote Service Call

RSC is the platform for distributed cooperation. It is coherently accessible on all logical nodes. The RSC interface is based on a set of objects that represent typical operating system primitives, i.e., requests, ports, storage, and accounts. RSC itself does not provide higher level objects like files or a directory, but provides the "building blocks" for making such objects accessible

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

and manageable in the heterogeneous distributed environment. The RSC programming interface is a set of operations, i.e., high-level language library functions, defined for the RSC objects. All operations are performed by a RSC worker that resides in the logical node and manages the RSC objects of this node as well as all communication required to access objects on another node.

Object sharing is the paradigm for distributed cooperation using RSC objects. To share an object (local or remote), access rights to the object are passed to the recipient who may then use the object in a way that is completely location, naming, and presentation transparent, i.e., just like a shared resource in a local system. Again, the local-system paradigm is the guideline behind the choice of RSC primitives and the object sharing. The system, i.e., the interacting RSC entities of DACNOS, provides the illusion of a single shared global object space.

As an example, consider a typical client-server interaction. The server program creates a port for its service and offers this port to certain clients. (*Create* and *offer* are RSC operations defined for the port object.) The port is typed in the same sense possible requests and request data formats are specified by the server programmer and attached to the port. It is thus the handle for an abstract service object. The actual data conversion is performed transparently by the presentation component when the data is sent across the network.

In order to bind to the service represented by a port a client will have to explicitly issue a RSC *share* operation. To send a service request to the server the client creates an object called *carrier* that specifies the request and also contains value and reference parameters. Reference parameters in RSC are passed as access rights to RSC objects. For example, access to a data buffer at the client side could be granted with the Carrier for the duration of the request. (Data buffers are described by *windows*.) The DACNOS data presentation syntax notation comprises mechanisms to specify reference parameters as part of an interface description [6]. The type of user data contained in RSC objects, e.g., carrier and window, is defined by an attached type description string. This is used by RSC to perform the necessary conversions. The programmer specifies the data types in a language that is an extension of ASN.1 [10]. This notation is compiled into a more efficient internal description string.

It is important to note here that the programmer of an application will only have to think in terms of application-related operations, e.g., create port, share port, call service, read data, while RSC transparently performs all the required communication, error handling, access checks, data segmentation, data conversion, and even account management [7]. Shared objects make distributed programming "look and feel" like local programming. More information on RSC including a detailed description of its objects and implementation can be found in [6].

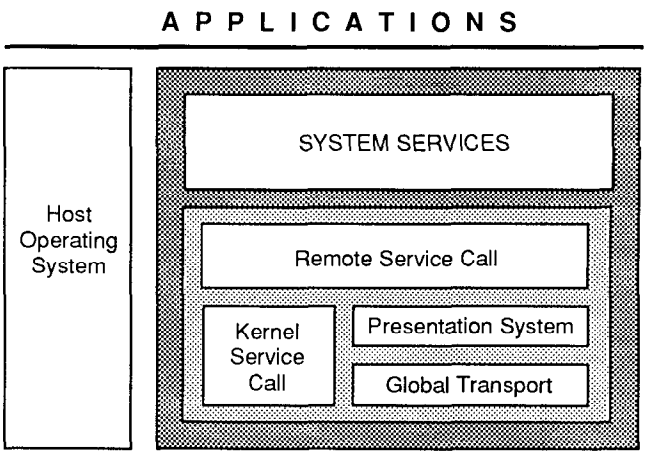

FIGURE 1.
DACNOS Architecture

Global Transport and Kernel Service Call

Global Transport (GT) and Kernel Service Call (KSC) are those components that serve the portability of RSC. The portation of RSC is basically the portation of GT and KSC to the target machine.

GT is the coherent interface for the data transport needs of RSC. The network is conceived as a set of interconnected islands of homogeneity with proprietary internal transport mechanisms. We did not impose a less efficient heavy-weight protocol where a specialized protocol performs much better. Thus, we use standardized protocols (OSI/TP-4, TCP/IP) only when a communication path crosses island boundaries.

GT basically is a reliable datagram interface to data transport between DACNOS logical nodes with a global OSI-style addressing scheme. RSC sees a simple "send-receive" interface where, for example, sequencing and duplication control are not required separately from the general error handling mechanisms in RSC. Internally, GT might very well use—depending on the transport protocol and network—connection-oriented services to send the datagrams between the network nodes. RSC, however, does not see connections on the transport level.

While GT handles diverse communication environments, KSC covers diverse host operating systems. Layered communication software typically deals with independent asynchronous events and, therefore, requires appropriate operating system support. With KSC we created an operating-system independent interface that offers communication–software-oriented services such as multiple light-weight processes, communication within shared memory, synchronization primitives for disjunctive multiple event handling, and timer services. Popular operating systems differ significantly in how much of this support is available and how it is offered to the applications. Consequently, the implementation efforts for KSC have ranged from "simple mapping of

services" to "implementation of a coexistent multitasking system." (See the section on "Implementation.")

It is interesting to note that KSC is based on the same cooperation philosophy as RSC: KSC processes cooperate through objects in shared local memory, and RSC processes cooperate through objects in a (virtually) shared global object space.

System Management Services

The DACNOS kernel provides communication, synchronization, and low-level access management support for application-to-application cooperation across a heterogeneous network. The DACNOS system management services complement these facilities with user- and resource-level services that help to organize and control the user access and resource allocation. These services are analogous to services in host operating systems and have to reside on nodes that are trustworthy.

The *Directory Service* maintains, distributes, and protects information about network resources and services. It also contains a name service that controls the naming of resources in the network. The DACNOS directory is not specifically tailored for a specific application scenario, e.g., for message handling or file stores, but aims to be a universal directory capable of supporting a wide range of applications. An entry in the directory database is simply a tuple consisting of object name, entry owner, and a set of type specific attributes. Access control is performed on a per attribute basis by owner-controlled access lists [15]. The name space is divided into domains of unique names. A server would register its service at the directory specifying some name string. The name service part of the directory ensures the uniqueness of the name within the server's domain. Clients find out about a service by asking the directory for a particular service name. The directory's response contains the network address of a server that offers the desired service. Using the service name and network address the client establishes a binder to share the server port (share operation). After the share has succeeded the client will in subsequent operations only use the port handle that is a result parameter of the Share Port operation [15].

The *Authentication and Authorization Service* (AAS) provides for mutual identification of interacting partners and is a means for servers to control access to their services. It is the key component needed to cope with the problem of non-secure, freely accessible workstations in the network. The authentication is based on a password scheme, and it is assumed that a user cannot forge his or her transport address. Users have to log-in with their AAS before using DACNOS services. Thus a "DACNOS user-id" is then correlated with a transport address. To check the authorization of a client, servers present to the AAS the network address and user-id as given in the client request in order to decide about the validity of the purported identity [14].

The DACNOS *Account Service* is analogous to the accounting facilities of a local system. Resource consumption data is collected within the RSC kernel and shipped to the Account Server. RSC has a special *account* object to support these functions. The data provides valuable information on the utilization of network services and can be used in billing for a provided service. For example, there are many ways to limit the consumption of resources by certain users or user groups. A broad discussion of the problems of accounting and billing in heterogeneous environments and the DACNOS Account Server can be found in [7].

All DACNOS system services described here are distributed in a sense that more than one server may be involved in order to serve a service request. The reader should refer to the cited references to get more information on server interaction and protocols. On top of the management services there are applications that are also considered DACNOS system services. These services, e.g., Remote File Access and Remote Execution, provide access to *shared resources*. They are discussed in the following section.

In a distributed system, more and new types of failures are possible than in the local case. DACNOS tries to assist the application programmer—as far as possible—in the handling of unexpected events. The representation of an invocation as a carrier makes the relations between remote cooperating components explicit to the RSC kernel. A watchdog process inside the RSC kernel periodically checks the availability of the remote partners by sending probe messages. Loss of connection to a node and loss of a service will thus be detected. Failures are reported to the waiters on a carrier or a port by terminating their wait-state with a special return code. It is then the responsibility of the application to react reasonably within its context. Mechanisms for the coordination of distributed transactions are known and could be integrated into RSC. Nevertheless, the applications would have to be written to support a transactional behavior.

APPLICATION EXPERIENCES

Two operating-system related services were developed partly in parallel with the DACNOS kernel: Remote File Access and Remote Execution. Both provided feedback on the design of RSC and its interface. This feedback helped to improve its functionality and interface style.

Remote File Access

Remote File Access (RFA) is a global, homogeneous file system for heterogeneous networks that provides transparent access to remote files [8]. According to our autonomy and transparency objectives, we do not replace any local file system, but accommodate the global RFA file system in the diverse local file systems. The RFA file system is partitioned into multiple RFA file servers, each being responsible for a subset of RFA files and RFA clients that mediate the user access to RFA. The running prototype supports sequential record-oriented files.

RFA servers use the local file systems of the host operating systems. This technique minimizes the effort needed to port RFA servers to different operating systems. It also allows existing local files to be made available globally without copying ("publish"), thus allowing easy exchange of files between RFA and the local file system. Published local files should not be modified without RFA; otherwise their global consistency can not be guaranteed. The RFA client software is an extension and in some cases a modification of the local operating system. The *extension* offers access to the global RFA files through procedure and command interfaces. The *modification* opens the local file system interfaces for the global RFA files. It intercepts calls to the file system and re-routes them if necessary. Sets of global files can be bound ("mounted") into the local file system as "virtual volumes." Global files can be accessed transparently via their local aliases in the same way as local files, and existing application programs may use global files without any change in the application code. This again is in line with our local-system paradigm.

The file naming structure of the global file system is hierarchical. Publication of a file at a file server includes assigning a global name to the published file. Similarly, binding a file into a local file system involves naming the file according to the local naming structure. Name mapping is assisted by user selectable default rules that can cover the most frequently used translations.

RFA uses RSC for cooperation between distributed RFA clients and servers. For example, an RFA file server offers a main port for "Open File" and maintenance-related requests. A private port together with a private server process is then created for each opened file. This port is shared with the client who uses it to access the file contents. These two reports differ in their visibility. The file server port is accessible to the public according to the defined access rights, yet an open file is a private matter between the client and the server. File data is exchanged between client and server using a shared RSC window object with the appropriate data description attached. Client and server only "read" or "write" to the arbitrarily sized window, while RSC handles the segmentation access control and the communication and conversion.

The mapping of open files to a port and a separate process has several advantages. The file server need not be concerned with dispatching. If multiple requests queue up, RSC contains the mechanisms to ensure a fair distribution of the file service among the clients. Authorization checking takes place during open time. For subsequent calls to the private port, the server can rely on the authorization of the caller. It is easy to add file service specific accounting to a server. Since the service providing process runs under the account of the client, this process can be charged for any desired account units. RSC and the accounting server collect the bills for the client process.

As to the implementation of RFA it is obvious that portability can only be limited since RFA client and server software anchor deep in the different host file systems—though they do not hinder the local functionality. Nevertheless, many RFA modules are portable based on the common programming language C and the RSC support. RFA clients with transparent remote file access are available for VM/CMS, VMS, and PC DOS. Designs for AIX and OS/2 have been done, but not yet implemented.

As mentioned before, RFA had influence on the RSC design. For example, a window's data format description originally was statically defined where the window was created. This was insufficient in cases where the client creates a data window in his virtual memory without knowing the actual record structure of the window that the server would use to write the file data.The server did not have means to specify data formats for the retrieved data whose structure was not known a priori to the client. This was changed in a way that one side can create a window with a wild-card data format description, which allows a sharer of the window to provide the structure information.

RSC has matured to become a powerful and convenient base for complex applications like RFA. Some of the advantages of using RSC for RFA are as follows. There is no need to design protocol elements for authorization, accounting, node failures, time out, or any other aspect of remote communication. This makes the interface design much easier and increases its stability. The data presentation functions of RSC are flexible enough to handle headers or trailers of variable length records transparent to RFA, i.e., without the need to reformat or mark retrieved data. Furthermore, the RSC window object is not only convenient, but also allows for transparent data transfer optimizations, e.g., a bulk transfer protocol is applied whenever it seems to be appropriate. This is transparent to the application that uses only *Read window* and *Write window* operations.

Remote Execution

The DACNOS Remote Execution Service (RES) [19] enables the sharing of programs located on remote computers. The aim was to build a natural and rather transparent extension to the invocation of local programs. Total transparency for remote execution is almost impossible to achieve in such an inherently heterogeneous environment, and we did not try to push the transparency limits. Rather, we wanted to provide a remote execution service that looks very familiar to the user.

Commonly, programs are designated by names. The user interface for starting a program has been extended such that the user can optionally append a server name to the program name where the program is to be executed. If no location is given with a program name the RES client part inquires at the directory service where this program is offered. The directories contain information about which programs are available on which

computers. The user can specify a directory search scope in order to limit the search to certain nodes or domains. If a potential location was found, the execution request is sent to the given RES server.

Executing programs will require additional input and will output results. File access of a remote program is handled by the DACNOS RFA component. Thus, the remote program runs with the current working file directory of the user that requested the execution. All terminal I/O is intercepted and forwarded to the client and server side, respectively.

RES places only simple requirements on the data presentation since terminal I/O contains characters only, and other data access is handled by RFA. During the development of RES, it was evident that the design and implementation of interactions between clients and servers based on the RSC platform (plus RFA) was almost trivial compared to the mastering of interceptions for terminal I/O and commands. These interception routines are obviously system dependent and not portable. Most of the rest of RES is portable. It has been implemented for VM/CMS, VMS, and PC-DOS (client only).

Database and Computation Server

To learn about the "ergonomics" of RSC and for demonstration purposes, remote access to an SQL database [23] and a high-precision numerical subroutine library were developed by summer students. In both cases, most of the design and all of the implementation was done by the students who had some programming experience but no DACNOS knowledge.

Within a few weeks after they had gained sufficient knowledge about the sub-systems to be accessed, they completed the implementation of a framework for the remote access using RSC objects with character data only and simulated access to the database or library. Without RSC this certainly would have required much more time in order to learn about the various interfaces, to develop the application and to debug it. Adding support for other data types took a matter of days. In both cases the client components were ported to all DACNOS systems. This was no effort at all, since the clients did not call any machine specific functions (only common C functions); and distributed cooperation was based on RSC.

IMPLEMENTATION

The design and implementation of DACNOS was a joint effort of researchers at the University of Karlsruhe and IBM. The project duration was limited to four years. All together it took an estimated 40 person-years to build the prototype as it is today.

All of our code has been written in the programming language C, except for some low-level KSC routines, which were better done in assembly language. Although the various implementations of C on heterogeneous machines have their compatibility problems, e.g., order of bit fields, alignment of structures and unions, default types, sign extension for shift operations, it was certainly the best available choice. It made our software highly portable, as long as certain conventions and rules were obeyed. A few numbers on the amount of code produced for the NOS kernel (excluding the DACNOS System Services) shall illustrate the development work: the KSC component (for VM/CMS) has about 7,000 lines of code, half C and half assembly language. The GT for VM/CMS consists of 6,500 lines of C code. Both figures vary depending on the host operating system. The NOS kernel, i.e., RSC including the data presentation, has roughly 55,000 lines of code and occupies 160K bytes of memory under VM/CMS.

Consequently, our NOS is an add-on to the different operating systems that does not interfere with existing applications, but makes it feasible to have access to remote resources in formerly only-local applications.

Host system dependencies of the RSC code are all completely separated into a collection of files, which have to be adapted when porting RSC.

KSC Implementation

As outlined in the section on "Architecture" KSC provides a coherent, communication-software-oriented surface on top of the host operating systems. It offers, among other facilities, light-weight processes sharing an address space. The portation of RSC is basically the portation/implementation of KSC. The amount of work varies depending on what is available in the host operating system.

For DACNOS on VM/CMS, each virtual machine (VM) is a "logical node" and represents an independent RSC entity with several internal and potentially many user-defined processes. We therefore had to add a transparent, coexistent, light-weight multitasking system to a VM, which originally did not offer support for multiple processes. It is important to note that KSC must not interfere with existing applications. Before KSC is added, the "CMS user process" is the only active thread in the VM. With KSC this view of the machine is still supported, but it is possible to create additional threads and thus multiplex the VM.

For VAX/VMS a logical node corresponds to a VMS process. With KSC this process can be split up into light-weight processes that share its address space. This implementation is analogous to the VM/CMS version, i.e., a VM in VM/CMS corresponds to a VMS process. The AIX version of KSC is also along this guideline [16] whereas OS/2 offers suitable facilities (light-weight multitasking with shared memory) that make the im-

plementation of KSC basically a functional mapping.

PC DOS was important for us as a widely available low-cost system. A PC was considered a single logical node in DACNOS. KSC was implemented by mapping the KSC process constructs onto a multitasking system that was internally available in IBM for the PC. Although this was relatively easy, the PC eventually gave us a hard time because of memory size restrictions and lack of memory protection. Though we succeeded to port DACNOS to PC DOS and implemented the transparent Remote File Access client in the PC file system, the applicability of DACNOS on a PC is rather limited due to the severe lack of memory. DACNOS plus the RFA client leave free less than 70K of the 640K main memory of a fully equipped PC. We did not investigate the use of extended memory under DOS, which might give some relief from the storage problem, but still does not cure the lack of protection. Since OS/2 and AIX became available and better exploits the increased power of workstations, we did not further invest into the PC DOS version of DACNOS.

Performance

Performance has always been a high priority design goal for DACNOS. (Highest priority was to find a *systematical* and *general* solution for heterogeneity in distributed systems.) Given the DACNOS constraints and network environment our design could not exploit some of the mechanisms and techniques that have frequently been used in other projects on distributed operating systems. For example, the DACNOS network may consist of a variety of interconnected local area networks and point-to-point lines with very diverse performance and reliability characteristics. The "cost" of a message is quite high. Therefore, extensive use of multicast or broadcast was impossible, and we tried to minimize the number of protocol messages without sacrificing functionality. Another example is the KSC, which is added on top of the host operating system. This obviously limits the KSC performance to the performance of the underlying general purpose system and is hardly comparable to an approach that builds a kernel directly on the bare hardware. These approaches make different assumptions and aim at different goals than DACNOS.

We measured the performance of selected RSC interactions involving various machines and communication links. The host measurements were taken during regular daytime use with light to medium load on the hosts. Some examples:

- The measured round trip time of an empty request between two VMs on an IBM 4361 was 50 milliseconds (msec). About the same time was observed on a VAX 8600.
- The same request between two VMs on two separate IBM/370 machines (4361 and 3083) connected by a 64K bits-per-second link using a proprietary protocol took approximately 125 msec, yet it took 210 msec on two VAXs (8600 and 8300) with VMS connected by an Ethernet and DECnet protocols.
- For two PC/AT personal computers on a token ring the empty request took 165 msec to complete, and PC to IBM 4361 host via token ring and gateway took 345 msec.

When more user data is sent with the request the execution time is the sum of the above-fixed amount for the empty request plus an increase proportional to the speed of the communication link.

We were also very much interested in the overhead introduced by the DACNOS kernel compared to the basic, unenhanced inter-process communication facilities of the host system. This is best expressed in the approximate number of machine instructions. For a round trip request the RSC client performs 6,000, the server side 8,000 instructions. GT (including KSC, but excluding the transport protocol itself) adds another 1,500 instructions for a send operation and 6,000 for receive. For the above-mentioned scenario with the client on an IBM 4361 (1.5 million instructions per second (Mips)) and a server on an IBM 3083 (5 Mips), this amounts to roughly 12 milliseconds for RSC and GT, a number we were quite satisfied with. If both client and server are located on the 4361 and only VM/CMS internal communication is used, the processing for RSC and GT takes roughly 20 msec and process switches, interrupt handling, and data copy operations, take the rest, i.e., 30 msec.

DISCUSSION OF RELATED WORK

In this section we restrict our discussion to related work that concentrates like DACNOS on adding solutions for heterogeneity and distribution to existing computing environments. We do not discuss distributed operating systems like Mach [1] or Locus [20], which can also work on heterogeneous hardware, but are built on the bare hardware. They are not meant to coexist with a local host operating system. Here we will discuss systems that are extensions to existing host operating systems.

Cooperation in heterogeneous distributed environments is facilitated by introducing a unified view onto the heterogeneity that is an abstraction from the given dissimilarities. Several locations for such an abstraction layer are conceivable. For the application programmer using DACNOS this abstraction is given at the RSC interface. (Other internal abstraction levels are the KSC and GT interfaces. These facilitate the portation of DACNOS, but are not seen by the "RSC programmer.") The RSC cooperation facilities, i.e., the RSC objects and operations, are coherently supported by all DACNOS nodes and were designed analogous to the facilities of a local operating system.

Another approach is to move the unification layer into certain "key" applications. In HCS [18] "key facilities," i.e., remote procedure call (RPC), naming and binding, are made compatible (or newly created, if not available) across the various types of systems. On top of these facilities common services are implemented that are considered most important: file store, mail, printing,

and remote computation. In HCS, heterogeneity is "accommodated" into certain services, but it is "abstracted" in DACNOS at the operating system level.

A project that shares the emphasis on heterogeneity and the need for a comprehensive solution with DACNOS is Cronus [21]. Object orientation, the integration of access protection into the kernel, a network independent transport interface, software portability, and operating system add-on rather than replacement are common properties for both systems. Though the design objectives are very similar, there are a number of differences in the approach and the implementation. For example, the Cronus object is on another level than the basic "building block" objects of RSC. There are object managers for each type of Cronus object while there is only one RSC entity that manages the objects of an RSC node. Cronus objects can be replicated and can migrate, which requires extensive use of group—and multicast search operations. Therefore, it is practically essential to have adequate communication support, i.e., a fast LAN with broadcast capabilities. RSC objects cannot be moved or replicated. The RSC protocol does not use broadcast or multicast facilities, although the DACNOS Directory and Orientation Service occasionally will have to issue search operations. Cooperation under RSC is based on the object sharing paradigm: the programmer thinks in terms of sharing objects just as in the local case, and message passing is the implicit mechanism used to implement such an "illusion" in the distributed environment. In Cronus, cooperation is achieved using explicit message-passing primitives and the interaction style is much more communication-oriented.

Sun RPC [24] and Apollo NCS [2] are representatives of Unix-based RPC packages. Both use the services of the underlying host operating system and transport service and support heterogeneity. In NCS, client and server stubs are generated automatically by a compiler from interface descriptions written in a "Network Interface Description Language." The Sun RPC only provides an extensible set of library routines for the marshalling and demarshalling of parameters. It is the responsibility of the application programmer to make the appropriate calls. In DACNOS, the invocation of remote operations is done by sending an RSC carrier. RSC, however, supports a different, more flexible and powerful interaction model. Requests are associated with carrier objects that may contain data values and object references as parameters (see the section on "Architecture"). Sending a carrier is an asynchronous operation. Therefore, clients may have several carriers outstanding and may selectively wait for their completion. Servers may receive and work on multiple requests simultaneously. The carrier also contains automatically appended information for system management purposes, e.g., authorization, accounting, and dispatching priorities [7]. In all three systems various services are built on top of these communication kernels. Their differences are not discussed here, because the major focus of this article is on the kernel of DACNOS.

There are also language-based approaches to conquer heterogeneity. In DAPHNE [13] coherence is achieved at the programming language level. Components of a program can easily be distributed for execution on different nodes of a heterogeneous network. The means for cooperation between the distributed components is a Remote Procedure Call (RPC) that is adapted to handle the heterogeneity. It is supported by a stub generator and appropriately modified run-time environment. In [5] a programming language (Network Command Language (NCL)) was defined for the description of remote service requests. Using pre-defined function libraries and additional server specific functions NCL expressions are created and shipped to a server. (A canonical data representation solves the data incompatibility problem.) With command language expressions, a client can "program" the server to perform a sequence of functions in a single request avoiding the overhead of multiple remote procedure calls.

Our main goal was to provide system level support that takes most of the burden of distribution and heterogeneity away from the programmer of a distributed application.

We would also like to mention two other very prominent attempts at mastering heterogeneity: Open Systems Interconnection (OSI) and IBM's Systems Application Architecture (SAA). OSI defines standards for the communication between heterogeneous computer systems [9]. The set of standards is still evolving. Only lately, efforts have been started to define a platform for distributed applications that goes beyond the mere communication aspects of distributed processing [11]. DACNOS is considered one of the prototypes for a support environment for ODP, which is being developed by the European Computer Manufacturers Association (ECMA) [4]. The ODP activities will produce a reference model about how to integrate and describe the various aspects of distributed computing like communication, directories, security and management. Such a framework, however, will not provide specifications for the implementation on a real system or portability considerations, i.e., problems that were solved in DACNOS and related research projects.

SAA is a software architecture for the development of consistent applications across the major IBM computing architectures [25]. SAA specifies common interfaces and conventions for user access, communication, and programming on dissimilar operating systems. Benefits of such an architecture will be easy migration between systems, the portability of software, and the elimination of redundant development efforts. DACNOS has tackled the subset of the SAA objectives, which are related to distributed processing in a slightly different, historically grown, mixed vendor computing environment. Never-

theless, the DACNOS prototype demonstrates not only the feasibility but also the potential benefits of SAA [22].

CONCLUSIONS AND OUTLOOK

The motivation for DACNOS stemmed from the demand for resource sharing in historically grown heterogeneous computer networks and the fundamental lack of appropriate support for such applications. Our main goal was to provide system level support that takes most of the burden of distribution and heterogeneity away from the programmer of a distributed application. The DACNOS prototype demonstrates the feasibility of such an approach.

Feedback from applications on DACNOS has provided us with valuable insights into our design decisions. The more general observations are: First, we have found that the "remote-like local" design principle makes the interfaces easy to comprehend for application programmers. This familiarity speeds up the development process and increases the productivity. Second, the integration of access and access management into the kernel relieves the programmer from explicitly dealing with much of the (in one way or another) required access management. Separating management functions and shifting them into the NOS kernel avoids costly "reinventions of the wheel." Third, the RSC object interface has proven to be a functionally complete "application-enabling" interface. This style of interface is not necessarily tied to the current implementation of DACNOS.

We could well imagine having such an enabler for distributed applications on top of other data transport environments, possibly implemented on a kind of host operating system support other than KSC. We claim that transport level primitives are too low-level for application programmers while the remote procedure call unnecessarily imposes a certain programming style that is often inadequate and sometimes cumbersome for the cooperation of independent networked processors. RSC is located somewhere in between the two offering an application-oriented abstraction that integrates communication as well as management functions. Finally, on a platformlike DACNOS application programs are potentially portable whether or not their scope is only local or remote. This clearly goes beyond the mere ability to communicate with other heterogeneous systems by using standardized communication protocols. DACNOS has most of the functionality that is being discussed in OSI standardization activities; plus, it presents a solution for system integration and software engineering problems.

Such a comprehensive support is not free. The complexity of the NOS kernel's design and implementation is higher than for approaches that run under the "keep it simple" mode. Less functionality in the kernel, however, tends to lead to replication of development efforts, less coherency between components, and thus, potentially reduced interoperability. Enforcing mechanisms in the kernel is the canonical approach with a higher complexity for the system designer whereas the noncanonical approach shifts some of the complexity to the application programmer with all the potential pitfalls—and some potential performance and optimization benefits. The DACNOS prototype, however, demonstrates that a functionally complete, more complex kernel can still perform very well.

The DACNOS development has taught us that in order to manage the complexity of a design effort it was extremely helpful to have a clear initial design guideline, i.e., in our case the *local-system paradigm*. This brought orientation in the early stages of the design process. It also brought consistency as we went along with the system design. And it helped to structure the intially huge problem space. No one does it completely right the first time. So it is almost needless to say that we would make several technical modifications and extensions if we had to do it again. None of these, however is related to our overall approach and design philosophy.

For example, we underestimated the problems of coordinating software development, versions, and maintenance on different computers by several people at different places. Usage of a software control system from the beginning of the project would have helped to eliminate several misunderstandings, incompatibilities and duplicated efforts caused by version-handling errors. We now think that CASE tools should be integrated into the development and implementation process. The implementation work of the DACNOS kernel was structured vertically, i.e., responsibilities were divided by the type of the computer system. Thus, the developers had to know internals of all kernel components. A horizontal structure would assign responsibility by component. This requires the developer to know several systems, but system interfaces are stable compared to the internals of software components under development. Therefore, a horizontal structure is potentially more efficient. KSC was designed to isolate local operating system dependencies from the rest of the DACNOS software. Full screen input and output to the user's terminal were not included. We underestimated its impact on the structure and portability of reapplications. Today, we would consider the integration of an existing window-oriented interface into KSC.

The development of DACNOS is basically finished. The prototype is in use at various locations outside of the IBM ENC. It was selected as a development basis for distributed applications because of its unique functionality and flexibility. Among the external users are two European universitities and an European RACE project. So far, user reaction is very positive.

Our experiences with usability and performance were confirmed. Currently, there are not enough users to publish statements on the scalability of our design. We intend to provide such data in a future report. There are still some activities going on to complete the portation of DACNOS System Services to the operating systems AIX and OS/2. We see opportunities for further extensions to DACNOS-like transactions, fault-

tolerance, support for distributed debugging, and support for distributed applications that incorporate multiple information media like data, voice, and video. It is unclear to us what kind of system support suits the programming of these applications and whether DACNOS can be useful as a starting point for such a system support. These questions will be the focus of our future research.

Acknowledgments. Many people, too numerous to mention, have contributed to the DACNOS project. Special thanks go to Hermann Schmutz for design contributions and management guidance and to Herbert Eberle for his design and invaluable implementation contributions. The following persons have contributed significant parts of the code of the DACNOS prototype: P.Brecht, H. V. Drachenfels, C. Föerster, G. Harter, A. Käemmer, B. Kieser, E. Kräemer, B. Mattes, S. Mengler, H. Moons, M. Philippsen, R. Öechsle, M. Salmony, B. Schöener, A. Schill, M. Seifert, P. Silberbusch, R. Staroste, M. Wasmund, G. Wild, H. Zöeller. We are grateful to all of them.

REFERENCES

1. Acetta, M., et.al., Mach: A new Kernel Foundation for UNIX development. In *Proceedings of Summer Usenix Conference* (Atlanta, Ga., June 9–13). USENIX Assoc., Berkeley, Calif., pp. 93–112.
2. Apollo Computer Inc. *Network Computing System Reference Manual.* Chelmsford, Mass., 1987.
3. Balkovich, E., Lerman, S., and Parmelee, R. P. Computing in higher education: The Athena Experience. *Commun. ACM 28,* 11 (Nov. 1985), 1214–1224.
4. ECMA Support Environment for Open Distributed Processing (SE-ODP). ECMA Tech. Rep., No. 49, Geneva, Switzerland, 1990.
5. Falcone, J. R. A programmable interface language for heterogeneous distributed systems. *ACM Trans. Comput. Syst. 5,* 4 (Nov. 1987), 330–351.
6. Geihs, K., et al. An architecture for the cooperation of heterogeneous operating systems. In *Proceedings of IEEE Computer Networking Symposium* (Washington, D.C., Apr. 11–13). IEEE, New York, 1988, pp. 300–312.
7. Harter G., and Geihs, K. An accounting service for heterogeneous distributed environments. In *Proceedings of the Eighth International Conference on Distributed Computing Systems* (San Jose, Calif., June 13–17). IEEE, New York, 1988, pp. 207–214.
8. Hollberg, U., Schmutz, H., and Silberbusch, P. Remote File Access: A distributed file system for heterogeneous networks. In *Proceedings of the GI/NTG Conference on Communication in Distributed Systems* (Aachen, West Germany, Feb. 16–20). Springer Verlag, New York, 1987, pp. 293–310.
9. ISO *OSI: Open Systems Interconnections Basic Reference Model.* International Standard 7498. Geneva, Switzerland, 1984.
10. ISO *Specification of Abstract Syntax Notation 1 (ASN.1).* International Standard 8824, Geneva, Switzerland, 1987.
11. ISO *ODP: Open Distributed Processing, Reference Model.* In preparation at working group ISO/IEC-JTC1-SC21-WG7, Geneva, Switzerland.
12. Krüeger G., and Müeller, G., (Eds.) *HECTOR.* Vol. I and II, Springer Verlag, New York, 1988.
13. Löehr, K. P., Müeller, J., and Nentwig, L. DAPHNE: Support for distributed applications programming in heterogeneous networks. In *Proceedings of Eighth International Conference on Distributed Computing Systems* (San Jose, Calif., June 13–17). IEEE, New York, 1988, pp. 63–71.
14. Mattes, B. Authentication and authorization in resource sharing networks. In *HECTOR,* G. Krüeger and G. Müeller, Eds. Vol. II, *Basic Projects.* Springer Verlag, New York, 1988, pp. 126–139.
15. Mattes B., and Drachenfels, H. V. Directory and orientation in heterogeneous networks. In *HECTOR,* G. Krüeger and G. Müeller, Eds. Vol. II, *Basic Projects.* Springer Verlag, New York, 1988, pp. 110–125.
16. Moons, H., Verbaeten, P., and Hollberg, U. Distributed computing in heterogeneous environments. In *Proceedings of European Unix Users Group (EUUG) Spring '90 Conference* (Munich, West Germany, April 23–27). 1990.
17. Morris, J. H., et al. Andrew: A distributed personal computing environment. *Commun. ACM 29,* 3 (Mar. 1986), 184–201.
18. Notkin, D., et al. Interconnecting heterogeneous computer systems. *Commun. ACM 31,* 3 (Mar. 1988), 258–273.
19. Öechsle, R. A remote execution service in a heterogeneous network. In *HECTOR,* G. Krüeger and G. Müeller, Eds. Vol. II, *Basic Projects.* Springer Verlag, New York, 1988, pp. 169–182.
20. Popek, G. J., and Walker, B. J. (Eds.) *The LOCUS Distributed System Architecture.* MIT Press, Cambridge, Mass., 1985.
21. Schantz, R. E., Thomas, R. H., and Bono, G. The architecture of the Cronus Distributed Operating System. In *Proceedings of the Sixth International Conference on Distributed Computing Systems* (Cambridge, Mass., Feb. 22–24). IEEE, New York, 1986, pp. 250–259.
22. Scherr, A. L. SAA Distributed Processing. *IBM Syst. J. 27,* 3 (1988), 370–383.
23. Schöener, B., and Kieser, B. Transparent database access in a network of heterogeneous systems. In *Proceedings of the GI/NTG Conference on Communication in Distributed Systems* (Stuttgart, West Germany, Feb. 22–24). Springer Verlag, New York, 1989, pp. 415–429.
24. Sun Microsystems *External Data Representation Reference Manual.* Mountain View, Calif., 1985.
25. Wheeler, E. F., and Ganek, A. G. Introduction to Systems Application Architecture. *IBM Syst. J. 27,* 3 (1988), 250–263.

CR Categories and Subject Descriptors: C.2.4 **[Computer-Communication Networks]**: Distributed Systems—*network operating systems*; D.4.6 **[Operating Systems]**: Security and Protection—*access controls*; D.4.7 **[Operating Systems]**: Organization and Design—*distributed systems*
General Terms: Design, Experimentation
Additional Key Words and Phrases: Heterogeneity, network file systems, portability, transparency

ABOUT THE AUTHORS:

KURT GEIHS is currently a research staff member in the Operating Systems Research Group of the IBM European Networking Center, Heidelberg, Germany. His current research interest include heterogeneous distributed systems, open distributed processing (ODP), and applications for high-speed networks.

ULF HOLLBERG is currently a research staff member in the Operating Systems Research Group of the IBM European Networking Center. His current research interests include heterogeneous distributed systems, distributed file systems, and application programming interfaces. Authors' Present Address: IBM European Networking Center, Tiergartenstrasse 8, D-6900 Heidelberg, Germany, GEIHS@HDIBM1.BITNET and HOLLBERG@DHDIBM1.BITNET.