The HetNOS Network Operating System: a tool for writing distributed applications*

ANTÔNIO MARINHO PILLA BARCELLOS
JOÃO FREDERICO LACAVA SCHRAMM
VALDIR ROSSI BELMONTE FILHO
CLÁUDIO FERNANDO RESIN GEYER

{marinho, schramm, belmonte, geyer}@inf.ufrgs.br

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
INSTITUTO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Av. Bento Gonçalves, 9500 Bloco IV - P.O. Box 15,064 CEP 91,501-970
Porto Alegre - RS - BRAZIL

ABSTRACT

The HetNOS network operating system is a set of software layers laid over "native" operating systems to provide a distributed programming platform. The environment is composed of the HetNOS shell command language and the system calls interface (accessed through a procedure library). In both levels of interaction with users, the set of machines integrated by HetNOS are seen as a distributed virtual machine.

The HetNOS command interpreter, namely *hsh*, implements most functions present in more traditional command interpreters. It is possible to spawn, monitor, and terminate processes in any host in the network like in the local case. The HetNOS distributed kernel uses a symbolic, global, location independent, process identification scheme. Distributed applications are split into sequential processes, which interact with each other by message exchange. There are neither connections nor ports, being the communication mechanism strongly influenced by the process identification scheme. This paper briefly describes the HetNOS software organization, presents the HetNOS environment for distributed programming, and then compares HetNOS with related work.

1. INTRODUCTION

There have been two main alternatives to develop distributed applications. The first one is to use distributed programming languages, which can be roughly divided into two categories: *distributed logical address space* languages and *shared logical address space* languages [BAL89]. The other alternative is to expand a familiar language (e.g., C) with network primitives for remote process communication and remote process creation.

A language may explicitly or implicitly present parallelism and distribution. In the *implicit model*, parallelism is usually detected by a special compiler. In the *explicit model*, the language offers constructs that allow programmers to convert a problem specification into a parallel or distributed program. Examples of languages with explicit parallelism are SR [AND93] and NIL [STR83].

Although a more natural form of expressing parallel algorithms, distributed languages do not seem to be widely accepted yet. This happens

* this work has been partially supported by CNPq and CAPES Brazilian research agencies.

probably more due to cultural rather than to technical reasons, as users often refuse to learn a new language. The investment required to train personnel is the main drawback of this option.

The other possibility is to use a conventional language extended with a network Application Program Interface (API), that is, a library of network primitives for remote process communication, such as *sockets* and TLI *(Transport Layer Interface)* [STE90]. The programmer continues to use his or her familiar language, just having to learn some new functions, as the syntax and semantics of the language are preserved. Another advantage of this approach is its superior performance, because these primitives are closely related to the underlying operating system, indeed being usually provided with it.

However, these API's are not very user-friendly, because they oblige the user to deal with low-level details, such as communication protocols and complex data structures (filled, e.g., with machine addresses and port numbers). In addition, users must identify the location of multiple processes in the network, and open several remote sessions to monitor their execution, frequently facing problems to debug distributed applications.

This work describes the main aspects of the HetNOS programming interface, as well as the principles of the system architecture. The purpose of HetNOS is to be an intermediate approach between the simplicity and performance of libraries and the abstraction exclusive of distributed languages, being an evolution of the former approach. HetNOS is based on a distributed kernel and a set of system servers, which offer network services for user programs through a class of high-level functions. The advantages of the environment include: (a) dynamic cooperation between tasks of different applications (which do not have to be started at the same time); (b) absence of pre-compiled or automatically generated code to be integrated; and (c) the ease of application development due to the familiar environment and primitives simplicity.

First we show the internal HetNOS architecture. Then, we describe the programming principles in HetNOS, covering communication paradigm and process management, with examples of typical primitives. After that, we present the environment available to application programmers. Finally, we compare HetNOS with related systems, such as PVM [GEI90b] and p4 [BUT92].

2. SYSTEM MODEL

HetNOS can be classified as a *network operating system* for two reasons: for being a set of software layers laid over a set of *native* operating systems [GEI90a], and because of the limited transparency degree offered to users [TAN85]. It should be noted, however, that HetNOS is not intended to act like a complete operating system. The *native* operating systems are not replaced by a new *integrated* one. Heterogeneity is (mildly) present in HetNOS as it allows cooperative processing among a diverse set of machines and, to some extent, operating systems.

The structure of a HetNOS node, illustrated in Figure 1, is divided into hierarchical layers. Each layer is composed of one or more modules, and each module is implemented as an independent heavy-weight process. Interprocess communication (IPC) is achieved by sending data through communication channels. This structured organization is more generic and portable, considering that it restricts the interaction between modules to their specified interfaces. However, if structuring makes the system more manageable and increases its portability, it also affects the overall system performance. Even in the local IPC, in the current version all user processes interact exclusively by message exchange, not sharing variables in physical memory.

We will now briefly discuss each layer and module in the HetNOS architecture.

2.1 Unix Kernel

The first layer is composed of centralized Unix operating systems [BAC86, LEF89] for common hardware platforms. Such systems provide basic services to their users, like local process management, exception handling, and I/O operations. Most hardware details are hidden in this layer.

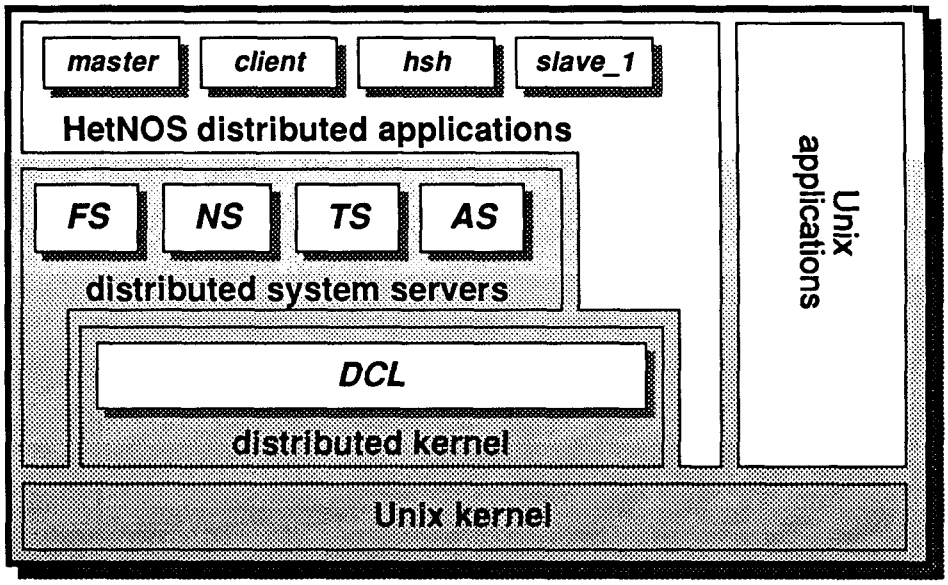

Figure 1 - layered local structure of a HetNOS node

2.2 Distributed Computing Layer, DCL

DCL is the distributed HetNOS kernel. It plays a fundamental role, implementing the distributed process abstraction (DCL hides process location from the higher layers). The main tasks assigned to DCL are:

- global, location-transparent, process identification scheme (also referred as *naming scheme*);
- interprocess communication mechanisms;
- distributed process management mechanisms;
- part of the security scheme (in cooperation with a server);
- dynamic and distributed network reconfiguration, covering ordinary node inclusion and exclusion, as well as failures in node or communication channels (along with other "specialized server").

2.3 Distributed System Servers

This layer implements higher-level services not found in DCL, thus complementing its functionality. There are four main distributed servers: **(a)** Name Server; **(b)** File Server; **(c)** Authorization Server; and **(d)** Type Server.

The **Name Server** (NS) acts like a generic global data repository, and was inspired in Linda [CAR89]. The information unit is the tuple, whose size and format are quite flexible. The interface consists of only four operations: inserting tuple, reading tuple, removing tuple, and modifying tuple. They correspond to the following primitives: `ns_put()`, `ns_read()`, `ns_get()` and `ns_change()`. The four primitives preserve mutual exclusion - in the case of conflicting operations, only one will succeed. The *change* operation was added (in relation to Linda) to cover the *one-writer-and-multiple-readers* applications.

Despite the scarce number of primitives, they are powerful enough to write 'logically centralized software with distributed hardware' applications [AND91]. Other HetNOS system servers

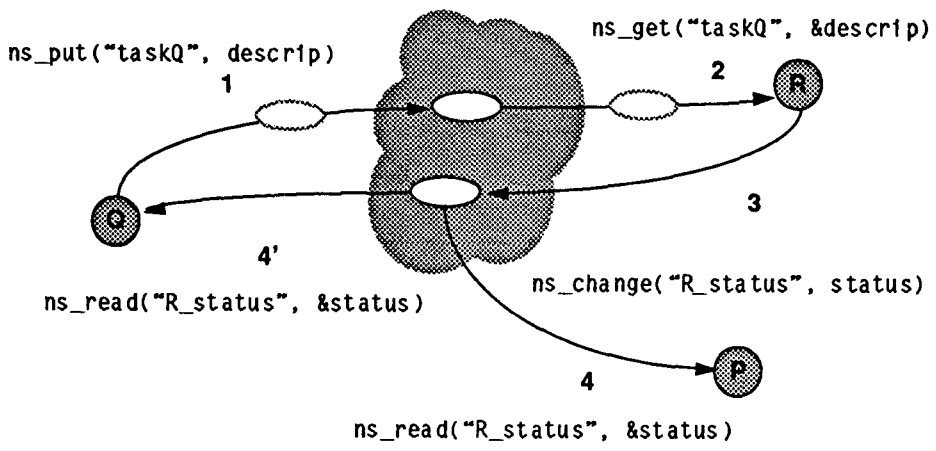

Figure 2 - Name Server tuple space

may, but do not have to, use NS services. In fact, currently, two servers have their implementations simplified by Name Server system calls. In the example of Figure 2, *P*, *Q*, and *R* are processes that may reside at different nodes. The tuple space is represented by the shaded area. First of all, (1) *Q* inserts a tuple. Then, (2) the tuple is read and removed by *R*, which then (3) changes a tuple in the space containing its own status. (4) This latter tuple is concurrently read by *P* and *Q*.

The **Authorization Server** (AS), along with DCL, controls the access to system resources. Checking user process permissions is one of the main AS responsibilities. These permissions are assigned by a privileged user, the HetNOS system administrator. The authorization scheme is based on the native operating system protection mechanisms (like DACNOS [GEI 90a]). To be able to access HetNOS services, a user process must have been created with HetNOS primitives (instead of the native Unix `fork()` system call). The Authorization Server does the accounting of system resources, offering statistical information about their use. In addition, AS also maintains static information about users (e.g., complete name), as well as miscellaneous information like date, time and local of user last login.

The **Type Server** (TS) performs copy operations of multiple instances of data structures between processes executing in machines with possibly incompatible architectures. The data types are defined through a special data description language, whose fundamental units are the types seen in the C language. The data being transferred is converted to a canonical type representation adopted by TS, and later, converted back in the target machine.

The **File Server** (FS), whose role and design are still under consideration, is responsible for the extension of the native file systems. It is aimed at integrating distinct remote file systems into a single global file system. FS should provide access to typed files (much like Saguaro [AND87]) in a transparent manner. A replicated multiversion file service is being designed [PED94].

2.4 Distributed Applications

HetNOS distributed applications are programs composed of multiple sequential processes communicating via explicit message passing. As previously mentioned, new processes are created by HetNOS primitives, which are similar to those found in Unix. Most Unix system calls can still be used without problems, as all HetNOS processes are implemented as Unix processes.

Some distributed applications have been implemented, such as fractal generation, Byzantine generals agreement, matrix multiplication, *queens*, and merge sort.

2.5 Global Organization

HetNOS global organization fits the *integrated model* [GOS92], and is similar to the structure typically found in *distributed operating systems* [TAN85]. Each HetNOS node runs a fully functional instance of the HetNOS operating system code. Each system module shown in Figure 1 interacts with its remote counterparts to provide upper layers with a *coherent* service in all machines. The resulting global organization is exemplified in Figure 3.

The original organization of HetNOS was a *logical ring*. To improve its performance and reliability, several changes have been made. The first major improvement regards network control - the inclusion of a ***Boot Server (BS)*** to monitor the ring. It allows super users to get updated information on the network (including the workloads) and change its configuration on-the-fly.

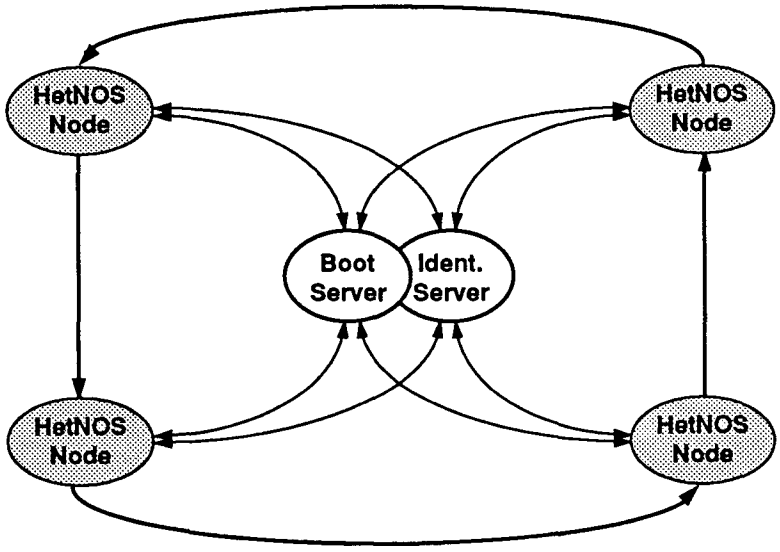

Figure 3 - Example of global HetNOS structure

The second major change was the inclusion of an ***Identification Server (IS)***. The role of IS is similar to a traditional Name Server, i.e., to identify and control object location. IS provides low-level services exclusively to DCL and system servers, while NS provides higher-level services accessed by both users and system modules.

In the current implementation, both the BS and IS run as centralized servers. To improve system reliability and scalability, we plan to implement distributed versions of both servers. Although using two centralized servers is certainly less elegant than using a fully-distributed model, in our case it was worthwhile due to performance gain. As can be drawn from Figure 3, all nodes (more specifically, the instances of DCL) are directly connected to these two servers.

The logical ring topology of HetNOS placed a too heavy burden on IPC performance. A non-local message was transmitted node by node up to its destination, and thereafter the message header would complete the round. To make IPC efficient a "fully-connected" scheme has been designed, and a first implementation added to the distributed kernel. This "fully connected" model is implemented using the UDP protocol (which does not employ transport-level connections). Messages can go directly from one host to another, without intermediate transfers. A *hint scheme* was used to learn process location, based on information firstly provided by the IS.

To join the ring, a new node first connects to BS and asks it permission to enter. Once granted, it is assigned a "left neighbor" to be contacted. BS interacts with the hosts involved, coordinating the operation. The DCL of the new node loads some global data from its left neighbor and begins the server initialization phase. Servers are started sequentially, in a pre-defined order, as a server may be implemented using services of

another server. At the end, BS broadcasts the new network configuration to all nodes.

3. PROGRAMMING MODEL

3.1 Messages as Communication Paradigm

One of the key aspects in the design of a distributed application is the interprocess communication. Among the most famous models, we can identify, in an ascending order of abstraction: *message passing* (MP), *remote procedure call* (RPC), and *distributed shared memory* (DSM). As a general rule, we can say that the greater the abstraction and sophistication, the poorer the system performance (although there have been improvements on DSM performance [ZHO92]). All these three options have their pros and cons [AND91, BAL89, GOS92, SIN94, ZHO92], and there is no agreement on which paradigm is the best one. This analysis is beyond the scope of this work.

One can interconnect Unix systems using a network API. The main problem with this approach is the need to handle host addresses, communication ports, etc. When using Unix sockets or TLI interfaces, one must choose the communication protocol, such as TCP and UDP [STE90]. TCP makes users deal with connections between processes, offering a reliable stream of bytes. UDP, on its turn, is message oriented but not reliable. With UDP, some messages may be lost, message contents may be corrupted, and message order is not guaranteed. This may force the user to implement his or her own verification mechanism.

Remote procedure call (e.g., Sun RPC/XDR [SUN90]) is a higher level IPC alternative. It is well suited to client-server applications, but it is not adequate for highly parallel applications, when there are lots of asynchronous interactions between multiple processes.

HetNOS adopts the message exchange paradigm, due to its flexibility and performance. It is possible to implement the two other paradigms using messages. Although being a relative low-level mechanism, this disadvantage is attenuated in HetNOS. It is not necessary to deal with the low-level aspects of communication (ports, protocols, locating and connecting to other hosts and processes). The naming scheme can be simple (e.g., static) or more complex (e.g., dynamic), depending on user needs and the complexity of the application. The goal is to achieve eventually the highest possible level of abstraction using the simple message exchange primitives.

HetNOS offers the flexibility not found on RPC, and frees the user from the job of managing message exchange that is associated to *sockets* or TLI. Processes send and receive messages using their identification, which is unique across the network. The target of a message is designated only by the name of one or more processes, regardless of their location.

All HetNOS primitives are reliable, but this does not imply the loss of message boundaries. An implementation of non reliable primitives is planned for performance purposes.

To simplify communication primitives, standard messages in HetNOS are not structured data types. Messages are "variable-sized packets containing a sequence of bytes ended by a *null character*" (a string in C). There are two possibilities to transmit data:

- formatting, sending and receiving a string with ASCII representations for all data types, using variable number of arguments routines, as in `printf("%d %s", i, s);`
- using special primitives to send and receive the message bytes unaltered (leaving to the

function the task of doing a byte-stuffing of the null character).

In general, the use of strings as messages results in greater simplicity and flexibility, compensating for the loss of performance this method incurs.

3.2 Process Management

Another crucial aspect related to distributed programming is process management. For example, when using *sockets*, TLI or RPC, processes are created using primitives like rexec(). The target machine must be specified, and the execution environment will be that of the remote machine, including name of the executable file, data files, search path, etc. The call is extremely inconvenient from a security point of view, considering that the user should usually either supply the password into the source code or prepare a configuration file containing the password. The user should also consider additional connections for both I/O and signal redirectioning.

In HetNOS, processes are created, listed, synchronized and removed considering the network of nodes as a "virtual machine". These operations follow the Unix philosophy, however in a simplified and distributed manner. When a process is created locally or in a remote host, it inherits all the important HetNOS and Unix attributes from the parent. The list includes associated tty, user identification, group identification, environment variables (e.g., *path*), etc.

Process identification plays a very important role in the implementation of distributed applications. In HetNOS, processes are identified with **names**, instead of integers. Names are labels, defined as a sequence of characters belonging to a subset of the ASCII code[1]. When users create a new process, they choose a name for it ("the baptism").

[1] 'A'..'Z', 'a'..'z', '0'..'9', '_' and '-'

Process identification is global to the network and location independent. For example, when a process executes the system call `h_kill("proc")` it does not have to know where the process "*proc*" resides. HetNOS searches the process and removes it just as it would be done in the local case.

A *location indicator suffix* is appended to the process name when it is necessary to explicitly identify the host where a process is being executed. This suffix is composed by the sign '@' and the name of the host (e.g., "*proc@host*" identifies the process *"proc"* at host "*host*"). Usually, only the system modules may need to employ such suffix (for instance, to address a specific instance of a server).

The global identification scheme allows communication between different applications, of different users, and in different machines. Considering that users are able to assign names to their processes, there could be name interference between identifiers of different users. Besides, there are names that are good candidates to identify processes of a distributed application, namely "server", "client", "coordinator", "calculator", "master", "slave", and so on. To solve this problem, the process name space is split in user-domains. The naming scheme uses an *owner identifier suffix*, which is an optional element (when omitted, names will refer to the user's own processes). The syntax is "*procname#username*", where '#' means ownership.

To illustrate, suppose a process named "wall" of a user named "floyd". To processes of user "floyd", the identifiers "wall" and "wall#floyd" refer to the same process. All other users may refer to this process using "wall#floyd". There are a few reserved names, corresponding to the names of HetNOS modules ("DCL", "FS", "NS", "TS", "AS", "BS", and "IS"), as well as "ANY" and "HetNOS". HetNOS does not allow users to

baptize their processes with one of these reserved names.

4. PRIMITIVES

Most arguments in the HetNOS interface are strings. Users might use C functions like `sprintf()` to set up more elaborated strings, when passing process names or messages as parameters. To help users, a whole set of HetNOS primitives was created to support calls with variable number of arguments. All functions that accept strings as parameters have their "_v" equivalent primitive. Similar to `sprintf()`, these system calls take first a format-description string, and then, according to such description, take a variable number of parameters to build the expected string argument.

4.1 IPC Primitives

In HetNOS there are several options of message passing primitives, most of them variations of the basic *send/receive* pair. All *send* primitives are *reliably blocking*, as the control is returned to the user only when an acknowledgment is received, indicating that the message has safely reached the destination host(s).

Regarding message **sending**, there are *synchronous* and *asynchronous* primitives. In the *synchronous send*, the source process is blocked until the message is read by the receiving process. In the *asynchronous send*, the source is freed even if the receiver is not ready to read the message (message is stored in a buffer in the destination host). There is also an *unbuffered asynchronous send* primitive: when the receiver is not ready to read the message, it is **discarded** (the message is not buffered) and an error code is returned to the sender. In the *multi-point send* the same message is sent to multiple process. There are *synchronous* and *asynchronous* variants of this primitive. In the former case, the sender is unblocked only when all targets have read the message, while in the latter, copies of the message are buffered in the target hosts and the sender may go on.

Regarding message **receiving**, there are only *synchronous* primitives. They can be *blocking*, *nonblocking* or *multi-point*. The *blocking receive* suspends process execution until an acceptable message is available. The *nonblocking receive* does not suspend the process, returning immediately either with a message read, or an error code indicating that the receiver would block. In the *multi-point receive* (also called *select)*, the receiver provides a process list, returning when the first message from one of these processes is available.

Finally, there is a pair of primitives dedicated to *asynchronous* transmissions of arrays of bytes, such as bitmaps. When processes are in homogeneous nodes, these primitives can also be used to send and receive data structures or program files.

A process may invoke the system calls `h_send()` or `h_sync_send()` to send data through a message (asynchronous or synchronous, respectively), such as:

```
h_sync_send(calcname, "let's sync");
h_send("talk_sv", "user connection request");
```

Additionally, a process can use one of the variable argument primitives to send an asynchronous message:

```
h_send_v(calcname, "(%d, %d) factor=%f",
   x, y, 1.5);
```

To receive a message, a process may execute the `h_receive()` primitive, providing the name of a source process ("ANY" can be used to designate acceptance from any process). The message sent in the previous example can be received as follows:

```
h_receive_v("ANY","%s: (%d, %d) factor=%f",
   &s[0], &z, &w, &f);
```

In all cases, the sender name can be safely obtained from the message itself, as HetNOS always inserts the name of the sender in the beginning of the message (along with a colon and a blank space). In the above example, the sender name is stored in the array of characters "s".

In some cases, it is more efficient or even necessary to use multi-point communication. In HetNOS, a process may send a message to a set of targets with a call like the following:

```
h_multi_send("P1 P2 P3", "to all of you");
```

A process willing to accept a message from one of the processes named "P1", "P2" or "P3" can use the h_multi_receive() system call, such as:

```
h_multi_receive("P1 P2 P3", &message[0]);
```

Arrays of bytes can be sent and received with, for example:

```
h_send_bytes("P1", &data_arr[0],
   sizeof(data_arr));
h_receive_bytes(source, &rec_arr[0],
   sizeof(rec_arr));
```

The last argument of h_send_bytes() is the number of bytes to be sent. The last argument of h_receive_bytes() is the maximum number of bytes the receiver wishes to accept.

4.2 Process Management Primitives

In the HetNOS programming model, applications should be partitioned in multiple processes, which will be distributed across the network. Process creation in Unix is based on two main primitives: fork(), which "duplicates" a process into a "parent" and a "child"; and exec(), that substitutes the current process image for another image [BAC86]. In this scheme, processes are always created locally.

As already mentioned, HetNOS "extends" the Unix process scheme. Users can make use of a "virtual machine" composed of several nodes running HetNOS. There is location transparency for all available process management primitives, as users do not have to bother about the location where processes are or will be created. However, HetNOS is a network operating system, and therefore it cannot access native operating system kernel tables, limiting to some extent the process management operations. As an example, there is not a process migration mechanism available.

The primitive for process creation by image duplication is h_fork(). Duplication is always local, as there is no migration capability. Ideally, processes should be created on lightly loaded machines.

In Unix, a fork() is frequently followed by an exec(), as this is the method to start a different program from another one. HetNOS offers three primitives for **both** creation and execution (*fork + exec*) of a program: h_loc_exec() locally creates and executes a process; h_rem_exec() creates and executes a process in a specific host; and h_exec() creates and executes a process in a host chosen by a load balancing policy. In the following example, a process starts a new child process (named "child") to execute the program "program" with arguments "arg1", "arg2", and "arg3":

```
h_exec("child", "program arg1 arg2 arg3");
```

As regards synchronization, a process may *rendezvous* with another process in two ways. Synchronization by means of messages can be achieved with the call h_wait_read(), which blocks the caller until all its messages have been read by a given receiving process. Alternatively, a process may issue a h_wait() call to wait for the death of another process (not necessarily one of its children).

Processes can be removed from the system at any time using the h_kill() primitive. Its behavior is not the same of the homonym Unix primitive. One difference is that h_kill() takes a list of processes to be eliminated. Another is that all

descendants of these processes are also always removed (regardless of their status, signal handling, or location).

For process self-termination, HetNOS offers the h_terminate() primitive. If a process terminates without calling h_terminate(), the HetNOS kernel detects this fact and takes all pertinent actions to ensure proper process termination.

5. ENVIRONMENT

HetNOS offers a simplified security scheme. Unfortunately, such scheme must rely on the security scheme of the native operating systems. A user is able to access HetNOS services only after a login procedure. New users are added to the HetNOS system by the HetNOS system administrator.

The HetNOS login application is hlogin, and is executed through a virtual console (e.g., a window of a native system graphical environment). This application asks the user for a *login* and a password. Then, this information is checked by HetNOS. If the information is authentic, a new session is opened. The user receives a greeting message and an instance of the command interpreter, hsh, is started.

Users submit primarily HetNOS commands to hsh, but can also run most Unix commands from it. The interpreter offers several facilities, common to other *shells*, such as environment variables, aliases, command history, and background execution. Some hsh commands are internal, like hkill, hps and hsend, which are used to eliminate processes, list existing processes, and send a message to a process, respectively.

Most communication primitives can be accessed by command line with hsh. Therefore, users can interact with his or her processes by sending and receiving messages from the command line, helping the debugging of distributed applications. The status of the process on the network can be continuously monitored.

The h_login() primitive opens a new session, while h_logout() closes it (this primitive is also called when the user enters the internal command hexit in the HetNOS shell). If a process successfully executes h_login(), then all HetNOS operations can be used by the process, as well as by its local and remote children.

Currently, there are no sophisticated tools for debugging in HetNOS. The monitoring of several process executions in the "virtual machine" is done with the hps command and automatic redirected output from created processes to hsh tty's.

6. RELATED WORK

Related work can be divided in two main categories: (a) distributed languages; and (b) distributed programming environments. An example of distributed language is SR (*Synchronizing Resources*), one of the best alternatives for distributed programming. There are several examples of distributed programming environments, such as CPS [RIN93], p4, and PVM. The main purpose of these environments is to exploit the computing power of idle workstations. Through the use of parallelism, it should be possible to carry out high-performance computing.

The SR language offers high-level mechanisms for the implementation of parallel algorithms. The basic element is the *resource*, which is similar to a module on other languages. The main advantage of SR over HetNOS is the paradigm of object oriented programming. The specification of a resource defines what objects can be exported and what objects can be imported from other resources. The message exchanging mechanisms (RPC and *rendezvous*) are easy to use and can be considered another advantage of SR.

On the other hand, HetNOS offers a more flexible distributed programming environment, with better process control mechanisms. A user does not have to define the location of objects, as happens in SR. Another advantage is application interaction. In HetNOS, different programs (executables) can communicate with each other, while in SR, interactions are restricted to tasks of only one program. A typical example is the implementation of a server that should service applications from different users.

HetNOS can be compared to distributed programming support libraries, as p4 and PVM. p4 is a library of functions and macros originally developed for parallel machines. Applications are written in C or FORTRAN and structured in processes communicating via message passing. A disadvantage of p4 is that communication is restricted to the modules of the same program. It is not possible for a program to send a message to another different program. Another disadvantage of p4 is the size of the executables (all files contain a copy of the communication system code). HetNOS executables are smaller, as much of the code resides in the distributed kernel (DCL), which is shared by all processes.

PVM (*Parallel Virtual Machine*) is a library supporting distributed programming for the C and FORTRAN languages. The processes composing an application are scattered on the virtual machine (set of machines treated as a single entity). On each node of this virtual machine there is a remote daemon. These daemons control local processes and communicate with other (remote) daemons. Both HetNOS and PVM associate unique global identifiers for each process. The identifiers in PVM are integers *(task id's)*, while in HetNOS they are labels (strings). The name of a HetNOS process may be chosen by the user, while in PVM the numeric process identifier is dynamically generated by the system (identifiers cannot be determined at compiling time). PVM process control is similar to that found in HetNOS, and includes operations such as: creating a process location independently or in a specific location, killing processes, listening active processes in the virtual machine, and even adding and removing hosts from the virtual machine. PVM defines the concept of *process group*, which is not available in HetNOS.

The PVM *send* primitive is asynchronous; synchronous communication is simulated with a *send* followed by a *blocking receive*. Much like HetNOS, PVM has *blocking* and *nonblocking receive* primitives, but does not have an *asynchronous receive*. HetNOS provides an additional *send* primitive, the *unbuffered asynchronous send*, which might be used, for instance, to distribute work load among a group of slave processes.

To compare the performance presented by the discussed systems, we implemented the *waves algorithm* [RAY90] for HetNOS, PVM, p4, and SR. We experimentally measured the process creation and the general IPC performance by executing the *waves* application in grids of 2x2, 3x3 and 4x4 nodes, having one process per host. The hardware configuration was an Ethernet network of Sun 4 workstations running SunOS 4.1.1. In general, the best performance was obtained running the PVM version of the application. p4, SR, and HetNOS presented similar performance on interprocess communication. Process creation was much more efficient in PVM and HetNOS than p4 and SR. The reason is that PVM and HetNOS use their own kernel to make process creation a local Unix operation in a remote host, while SR and p4 use the Unix rexec() service to create remote processes.

7. CONCLUSION

This paper describes the HetNOS basic design principles and programming environment. The main design aspects were discussed, specially

those related to the distributed kernel. Particular emphasis was put on topics like the modular and distributed system organization, the global process naming scheme, and the location-transparent interprocess communication paradigm.

HetNOS is a tool suitable for distributed programming in the Unix operating system environment using the C language. Applications are divided into processes - which constitute the basic execution unit - that are executed over a distributed virtual machine. These processes, which may not belong to the same application, communicate via high-level message passing primitives. Processes are identified by user chosen names, instead of numerical values assigned by the operating system. This allows users to assign a name to a process even at compiling time.

HetNOS prototype implementation consists of over 40,000 lines of C source code. The communication mechanism is presently based on the BSD *socket* API. The prototype was originally developed for Sun SPARC RISC platforms, but has been ported to a few others, such as Sillicon Graphics Indigo, HP Apollo and DEC 5000. The distributed kernel here described has been fully implemented to the SunOS with Sun/4 hardware, and is being ported to other platforms. Regarding the servers, there are fully functional versions of the Name Server and Authorization Server, while initial versions of the Type Server and File Server are under development. All these modules will have to be improved (to increase their overall reliability and performance) before a stable version of HetNOS is released.

As future work, IPC performance should be further improved, while new primitives should be made available. In the global context, system internal organization is being changed to take real advantage of the direct host-to-host communication that recently became available. In the local context, we are considering the reimplementation of the local message passing mechanism using shared memory IPC.

Regarding the HetNOS environment, a graphical interface is being designed. Its first part will treat process management in a graphic manner. It will be made of two main windows: program manager (containing executable files) and process manager (containing nodes of the virtual machine with executing processes). Users will be able to create processes by dragging executables onto iconized nodes.

Finally, we plan several improvements in HetNOS: (a) extend the process management scheme to support signaling and process groups; (b) add communication primitives similar to those found in SR, with typed message passing; and (c) add regular expression support to the HetNOS naming scheme, allowing users to issue commands or system calls such as "`hkill slaves*`".

8. REFERENCES

[AND87] ANDREWS, G.R.; SCHLICHTING, R.D.; HAYES, R.; PURDIN, T.D.M. The Design of the Saguaro Distributed Operating System. **IEEE Transactions on Software Engineering**, New York, v.SE-13, n.1, p.104-118, Jan. 1987.

[AND91] ANDREWS, G.R. Paradigms for Process Interaction in Distributed Programs. **ACM Computing Surveys**, New York, v.23, n.1, Mar. 1991.

[AND93] ANDREWS, G.R.; OLSSON, R.A. **The SR Programming Language: concurrency in practice.** Benjamin/Cummings, Redwood, 1993. 344p.

[BAC86] BACH, M. **The Design of the Unix Operating System.** Prentice-Hall, Englewood Cliffs, 1986. 471p.

[BAL89] BAL, H.; STEINER, J.; TANENBAUM, A. Programming Languages for Distributed Computing Systems. **ACM Computing Surveys**, New York, v.21, n.3, Sep. 1989.

[BAR93a] BARCELLOS, A.M.P. O Sistema Operacional de Rede Heterogêneo HetNOS. CPGCC-UFRGS: **MSc Thesis**, Porto Alegre, April 1993. 213p.

[BAR93b] BARCELLOS, A.M.P. Projeto do Sistema Operacional de Rede Heterogêneo HetNOS. In.: Seminário Integrado de Software e Hardware, 20 (XX SEMISH). **Proceedings.** Florianópolis, SC. Sept. 1993. Rio de Janeiro, SBC, 1993. p.718-732

[BAR94] BARCELLOS, A.M.P.; SCHRAMM, J.F.L.; TEIXEIRA JR., C.A.; GERHARDT, G.; GEYER, C.F.R. Um Ambiente para Programação de Aplicações Distribuídas em Redes de Workstations. In: Congresso Nacional de Redes de Computadores, 12 (XII SBRC). **Proceedings.** Curitiba, PR. May 1994. Rio de Janeiro, SBC, 1994.

[BUT92] BUTLER, R.; LUSK, E. **User's Guide to the p4 Programming System.** Argonne National Laboratory, Oct. 1992. 37p.

[CAR89] CARRIERO, N.; GELERNTER, D. Linda in Context. **Communications of the ACM**, New York, v.32, n.4, p.444-458, Apr. 1989.

[GEI90a] GEIHS, K.; HOLLBERG, U. Retrospective on DACNOS. **Communications of the ACM**, New York, v.33, n.4, p.439-448. Apr. 1990.

[GEI90b] GEIST, A. et alli. **PVM 3 User's Guide and Reference Manual.** Oak Ridge National Laboratory, May 1993. 108p.

[GOS92] GOSCINSKI, A. **Distributed Operating Systems: the logical design.** Addison-Wesley, Sydney, 1992. 913p.

[LEF89] LEFFLER, S.J.; McKUSICK, M.K.; KARELS, M.J.; QUARTERMAN, J.S. **The Design and Implementation of the 4.3BSD Unix Operating System.** Addison-Wesley, Reading, 1989. 471p.

[PED94] PEDONE, F.L.; BARCELLOS, A.M.P.; GEYER, C.F.R. Um Servidor de Arquivos Multiversão Replicados. XX CLEI Latino-American Conference. ***(submitted)***. Sept. 1994.

[RAY90] RAYNAL, M.; HELARY, J.M. **Synchronization and Control of Distributed Systems and Programs.** John Wiley & Sons, Chichester, 1990. 124p.

[RIN93] RINALDO, F.J.; FAUSEY, M.R. Event Reconstruction in High-Energy Physics. **Computer**, Los Alamitos, v.26, n.6, Jun. 93.

[SIN94] SINHAL, M; SHIVARATRI, N.G. **Advanced Concepts in Operating Systems.** McGraw-Hill, New York, 1994.

[STE90] STEVENS, W.R. **Unix Network Programming.** Prentice-Hall, Englewood Cliffs, 1990. 772p.

[STR83] STROM, R.E.; YEMINI, S. NIL: An Integrated Language and System for Distributed Programming. **ACM SIGPLAN Notes**, v.18, n.6 June 1983

[SUN90] SUN Microsystems, Inc. **Network Programming Manual.** Revision A, 1990. 297p.

[TAN85] TANENBAUM, A.S.; van RENESSE, R. Distributed Operating Systems. **Computing Surveys**, New York, v.17, n.4, p.419-470, Dec. 1985.

[ZHO92] ZHOU, S.; STUMM, M.; LI, K.; WORTMAN, D. Heterogeneous Distributed Shared Memory. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.3, n.5, p.540-554, Sept. 1992.

9. APPENDIX

As an example of HetNOS programming style, this appendix shows simplified skeletons of a client and a *concurrent* server [STE90]. It is concurrent because each client request received is concurrently handled by a child server created exclusively to satisfy the request. In the main loop, when a message is received, the server creates a child and goes back to wait for another request. The server defines the name of its child servers by joining its own name and the name of the client that requested the operation. Exemplifying, if the server is identified as "fooserv" and the client, as "fooclient", the child server created will be named "fooserv_fooclient". It must be noted that it is the child server that responds to the client.

The only thing to comment about the client is that the receiving operation is not specific ("ANY"), because, as previously mentioned, the answer to service is not sent by the server, but by its child. This solution assumes that the client will not receive messages from any other processes, apart from the server, after sending the request and receiving a reply. There are two alternatives if this condition cannot be kept: (a) modify the server to make the parent server itself send the reply; (b) make the client know how the server chooses the names of the child servers.

```
int
main()        /* CLIENT CODE */
{
   char       msg[MAX_MSG_LEN],
              sender[MAX_NAME_LEN];
   int        request_code,
              reply_code;
   h_init();
   ...
   h_send_v("dummy_server", "%d",
      request_code);
   h_receive_v("ANY", "%s: %d", sender,
      &reply_code);
   ...
   h_terminate();
}
```

```
int
main()        /* SERVER CODE */
{
   char msg[MAX_MSG_LEN], client_name[MAX_NAME_LEN];
   int  request_code;

   h_init();
   my_name = whois;/* my global process name */
   while (TRUE) {
       /* get new request from client and parse it*/
       h_receive("ANY", msg);
       sscanf(msg,"%s: %d",client_name,&request_code);
       /* create concurrent server to service
          request - name of the concurrent
          server is taken from the combination
          of server name and client name */
       if (h_fork_v("%s_%s",my_name,client_name)) {
            /* parent: do nothing - don't wait */
       } else {
            /* child: go servicing request */
            switch(request_code) {
            ...
            }
            /* reply to client */
            h_sync_send(source, reply);
            h_terminate();   /* and die */
       }
   }
}
```