A Distributed Interoperable Multi-Personality Environment

Robert C. Chen
William C. Davenport, Jr.
Christopher G. Kaler

April 13, 1992

Digital Equipment Corporation
110 Spit Brook Road
Nashua, NH 03062-2698
{rchen,davenport,kaler} @ star.enet.dec.com

Abstract

There are currently several notable and popular operating system environments, each with its own advantages and disadvantages. While each environment has its proponents, for many users the availability of applications dictates the choice of environment. Often, no single environment supports all desired applications, so users must either utilize multiple environments or forgo certain applications. Making multiple environments available simultaneously on the same system would help alleviate this situation. OSF and mt Xinu have recently demonstrated multiple environments executing simultaneously on top of a microkernel [13]. Our own recent efforts [12] led to a model of VMS[1] that separates its command language and application environment from the process management tasks. This separation helped us conceive a model that would support, for example, VMS and BSD UNIX[2] processes cooperating as components of the same system.

These developments led us to Intrepid, a model that encourages interoperability between multiple personality environments. Intrepid provides a framework that allows discrete pieces of diverse environments to work together in any combination. The Intrepid model supports a distributed environment that, for instance, enables the user to issue the command:

```
directory -size -date | more
```

This command is issued to a C-shell (*csh*) command interpreter that links the flow of input and output between the indicated commands (*i.e.,* sets up a pipe). The command invokes the VMS `directory` application to obtain a listing of files, the output of which is passed into the BSD program `more` for formatting. The output from `more` is in turn delivered back to the user.

The user in the above example might be logged into a DECstation (which is where the command interpreter is executing), while the VMS `directory` program is running on a Digital Alpha system, the BSD `more` program is running on an Intel 80486[3] system, and the user's files are located in a distributed file system spread over several hardware platforms.

In this paper we discuss the design of Intrepid and the supporting prototype work we have completed. We also present initial information on the performance of our prototype. Finally, we list our conclusions and suggest areas for future work.

[1]VMS, VAX, DECstation, and Digital are trademarks of Digital Equipment Corporation.

[2]UNIX and Open Look are registered trademarks of UNIX System Laboratories, Inc.

[3]Intel and 80486 are trademarks of Intel Corporation.

1 Motivation

The successful operating system of the future will be the one that garners the most applications, and this in turn will be the one that provides the best environment to application developers and their end users. Our goal with Intrepid is to create such an environment.

The Static View: Interoperation

Not only do different application developers prefer different operating system environments, but any one developer may prefer different environments during different phases of development. While working in one environment, a developer may wish to use a command language construct, call on a utility (say LSE on VMS, or yacc on UNIX), or call on services provided by a library of routines available in a different environment.

Intrepid is designed to fulfill these wishes by supporting interoperation among all popular environments. For example, a developer is able to issue the command `directory -size -date | more`, which combines the VMS command `directory` (with options `size` and `date`, to list the contents of a directory) with the UNIX command `more`. A developer is also able to issue calls to VMS library routines from a program being developed under MS-DOS[1]. Thus a developer can work mainly under her/his favorite environment, yet still take advantage of utilities, commands, and system calls that are available in other environments, invoking them in a perfectly natural way.

The Dynamic View: Adaptability

Popular operating systems such as VMS, UNIX, and MS-DOS will continue to evolve new facilities, and new applications will continue to be developed on them. In addition, new operating systems will be invented, incorporating new facilities, some of which will become popular. Intrepid needs to be sufficiently agile to support these new facilities, both to keep up with evolving operating system environments and to support new ones. A micro-kernel based multi-server approach helps provide this agility.

Intrepid also needs to be easy to port onto significantly improved architectures as they become available. At the same time, Intrepid users will want to continue using existing machines to support older applications. Consequently, Intrepid needs to support heterogeneous distributed systems. This indicates an approach where architecture-specific portions of Intrepid are clearly delineated so that they can be minimized and isolated.

The Distributed View: Cooperation

Intrepid provides its interoperable multi-personality environment over a *workgroup*. An Intrepid workgroup is a set of heterogeneous, loosely-coupled systems. The systems that constitute a workgroup support the sharing of their resources throughout the workgroup. Most servers in Intrepid can execute on any system within the workgroup.

Each workgroup exists and interoperates with other Intrepid workgroups. Each workgroup can also coexist and interoperate with conventional systems in more loosely-coupled networks.

The Vision

Intrepid provides a base environment that supports the natural intermixing of multiple programming environments and command languages. Applications are transparently available from all command languages. In turn, applications can make use of services provided by any supported environment. Intrepid treats environments and their underlying hardware as a set of resources that provide application availability. Intrepid is structured to allow the easy addition of new environments and command languages. It

[1]MS-DOS is a registered trademark of Microsoft Corporation.

exploits distributed processing, especially in regard to services provided by specific machines, and supports redundant services that allow the distribution of workloads and support for application fail-over. This is our vision of seamless interoperability.

2 Background

We recently developed a model and prototype of VMS that uses micro-kernel and client-server technologies. We wanted to cleanly partition VMS into subsystems and to explore architectural independence. The model supports multiple operating system environments sharing a common set of operating system servers. Our prototype system, VMS-on-Mach, implements a subset of this model on top of the Mach [1,6,10] micro-kernel. The ideas presented in this paper build on the foundations created by this work.

We will refer to each operating system environment as a personality environment. VMS-on-Mach (seen in Figure 1) creates a VMS personality environment out of the services provided by a set of generic system servers. Bootstrap of this environment is controlled by the executive server, which is also responsible for registration of individual servers and processes. Each process is managed by a process server that provides an address space within which process-protected code and data reside.

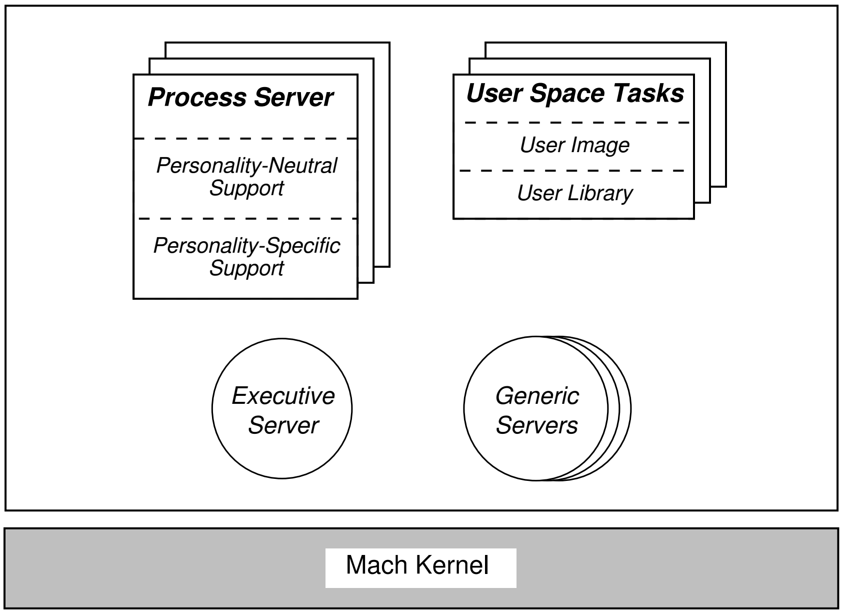

Figure 1 VMS-on-Mach Model

The process server provides personality-neutral and personality-specific services. Personality-neutral services (*e.g.*, process login and authentication, process state management) provide a framework within which personality-specific functions (*e.g.,* command language interpretation, image file activation, and application support services) are implemented.

User space tasks, which are created by the process servers, provide the address spaces within which VMS programs execute. Calls to the VMS system services [7] are handled by a read-only user library residing in each user space task. These service requests are implemented either directly by the user library or using functions provided by the set of generic system servers.

Generic system servers provide personality-neutral services that the process server and user space task use to create the VMS personality environment. These servers include an authorization server for validating a user to the system, an I/O server for input and output, a file-mapped memory server for VMS image activation, and a logical name server that provides a string translation facility for VMS logical names.

3 Intrepid System Model

The VMS-on-Mach model divides the process server, dedicated to a user, into personality-neutral and personality-specific components. Except for the personality-specific components of the process server and user space tasks, VMS-on-Mach is personality-neutral.

The natural extension of this approach to implementing multiple simultaneous personality environments is to supply a variety of versions of the personality-specific components. Instead, in Intrepid we take the approach of implementing a series of servers. Each server supplies services appropriate to some subset of the personality environments. For example, one command language server supplies the semantics for the Digital Command Language (DCL) [3] while another server supplies the semantics for the UNIX C-shell (*csh*) [11].

The Intrepid model (seen in Figure 2) is derived from VMS-on-Mach by partitioning the process server into a set of personality-neutral and personality-specific servers. This results in several new servers that allow us to provide a fully interoperable environment. This new model has the following features:

- An *Executive Server* that manages the complete set of servers within the system.
- A *Session Manager* that manages all user sessions (processes) present on the system.
- An *Application Manager* that is responsible for activating and managing applications. It coordinates the actions of other servers to bring applications into memory and to create the personality environments necessary for their execution.
- A set of *Personality Managers*, each of which implements a specific application environment (*e.g.*, BSD, VMS, MS-DOS). Associated with each personality manager is a *Personality Library*. This personality library resides within each application's address space to provide personality-specific services (*e.g.*,VMS system services) to the application.
- A set of *Image Servers*, each of which understands a specific on-disk image-file format and is capable of activating these image files into the address space of an application.
- A set of *Command Interpreters*, each of which implements a specific command language (*e.g.*, DCL, *csh*, *etc.*) for the users of the system.
- A set of *Generic Servers* that provide personality-neutral services (*e.g.,* file system, I/O, authorization, *etc.*) that are utilized by personality managers and libraries to implement specific personality environments.

User sessions are managed by the session manager, which is responsible for validating each user of the system. Once access to the system is granted, the session manager assumes a background role, and the user interacts directly with the selected command interpreter to process commands. These command interpreters, in turn, use the application manager to execute programs requested by the user. The application manager makes use of image servers to activate the application images and personality managers to provide the specific personality environments required by the application.

Intrepid partitions the traditional process into four entities: the session manager, a command interpreter, an application manager, and an environment-specific personality manager. In order to support seamless interoperability, each component must not only be independent, but usable from every other component. This is achieved by using personality-neutral interfaces throughout: all image servers provide a common set of services, as do all command interpreters and personality managers. This allows the application manager, for example, to communicate with any image server without needing to differentiate between image servers. Consequently, command interpreters can be combined with image servers and personality managers in many combinations to satisfy any requirements of a user or application. The ability to run any application from any command language promotes the sharing of resources in a distributed environment. The design also facilitates on-the-fly upgrades since new versions of these components can coexist alongside existing versions.

The following sections describe the major components of Intrepid in further detail.

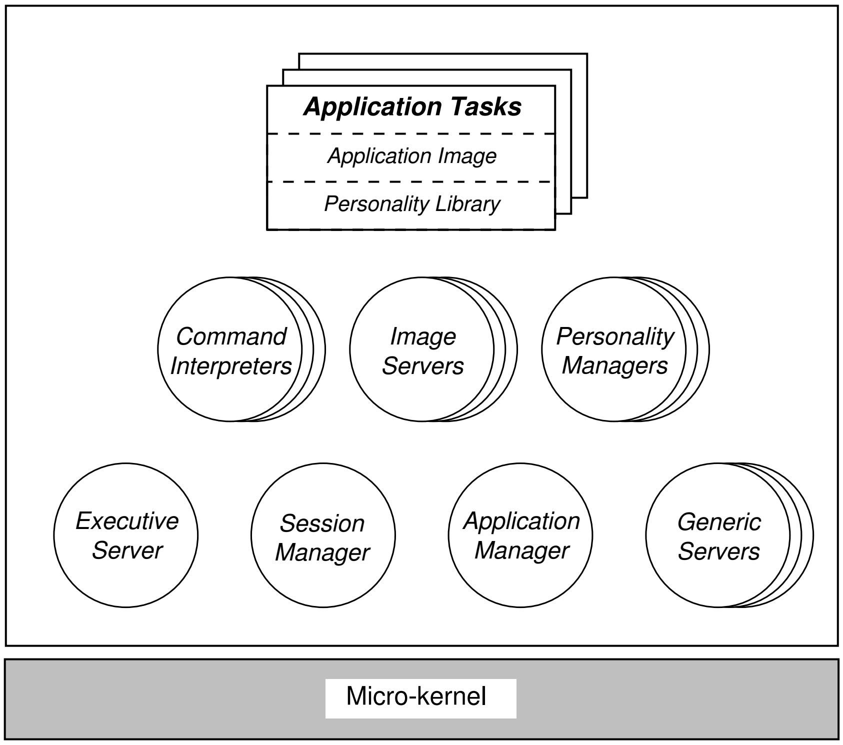

Figure 2 Intrepid System Model

Executive Server

The executive server manages all system-wide control functions. It provides bootstrap services for the set of servers present in the system, and functions as the system-wide name server for these servers. All servers register with the executive server prior to accepting and processing client requests.

Session Manager

The session manager is responsible for all user sessions associated with the system. It performs user validation, manages session-wide data (*e.g.,* user name) associated with each user, and serves as the focal point for session-related service requests (*e.g.*, session queries).

The session manager creates a session in response to a request from the user. For instance, a new session is created when notification is received from the I/O server that a user has begun input on an idle terminal. This new session (a thread in the session manager) uses the I/O server to request user login information, which it then asks an authorization server to validate. This validation process remains unchanged from the VMS-on-Mach model and supports a single system management domain that exists *outside* any specific personality environment. Once it has created and authorized the session, the session manager requests the selected command interpreter to begin command processing for the user. The session is terminated and its associated resources are reclaimed when the command interpreter informs the session manager that it has completed processing all commands for the session (*i.e.,* the user has issued a logout command).

Command Interpreters

Each command interpreter supports a specific command language (*e.g.*, DCL or *csh*). Commands that invoke applications generate requests to the application manager to activate the application. With this interface, command interpreters are not concerned with either the format of the application image file or the personality environment required by the application.

Since each command interpreter is an independent self-contained server, it can be utilized by other command interpreters to perform user requested actions. For example, a VMS DCL command procedure can be invoked from a *csh* command interpreter by its requesting the DCL command interpreter to perform processing of the command file.

Application Manager

The application manager processes user requests to invoke applications. It creates an address space for use by the application (the application task), and coordinates the actions of image servers and personality managers to activate the application and initialize the personality environment necessary for execution of the application. This design effectively separates the on-disk image-file format of an application from the personality environment needed for its execution. For example, an application that uses the VMS personality manager could be built using BSD tools and stored on disk in the UNIX *a.out* [11] format instead of the VMS *.EXE* [2] format. This application could then be run directly by a user from either the DCL or *csh* command interpreter.

When a user invokes an application, a request is sent to the application manager. The application manager determines the appropriate image server to load the application and the personality managers required by the application. Application requests for shareable libraries are returned to the application manager for image activation. By implementing a cache, the application manager can keep pre-activated applications and shareable libraries available for future invocations since all requests for image activation are directed to the application manager.

Personality Managers

Application support is the function of personality managers. Each personality manager implements a specific personality environment (*e.g.,* BSD, VMS, MS-DOS) for use by applications. Personality-specific programming interfaces (*e.g.,* BSD system calls, VMS system services) are delivered to an application by a personality library. This personality library resides within the application's address space and provides a translation of the personality-specific programming interfaces into requests to the servers present in the system.

The application manager uses personality managers to provide the environment required by an application. For example, the application manager would use the VMS personality manager to support programs that require the VMS system service interface and the BSD personality manager to support programs that require the BSD system call interface.

If an application requires both VMS and BSD interfaces, the application manager interacts with both personality managers and coordinates their interaction. Our belief is that an application should be able to make use of multiple programming environments. It is not clear, however, whether an application can have multiple environments active simultaneously. The extent of interaction between environments is defined by the personality managers and their personality libraries. Note that if servers are used to implement many of the underlying features provided by the environments (*e.g.,* I/O), the potential for interaction and interoperability is significantly increased. Interaction between these environments is limited by the amount of information that is privately maintained by each environment and is thus not available to other environments associated with the application.

Image Servers

Each image server provides support for a specific on-disk image-file format and is capable of activating these image files into the address space of an application.

Upon a request to run an application, the application manager creates an application task and requests one or more image servers to activate the requested program into the application task's address space. Since this requires knowledge of the on-disk representation of the application, each on-disk format is serviced by a separate server. That is, the VMSEXE image server handles requests for applications stored on disk using the VMS image-file format, the AOUT image server handles requests for applications stored in the UNIX a.out image-file format, and the COFF image server handles requests for applications stored in COFF image-file format. This separation allows applications to effectively utilize multiple on-disk image-file formats. For example, an application stored as a VMS image file might reference a library that is contained in an a.out image file. The various image servers, under the direction of the application manager, activate all components of the requested application into the application's address space.

Application Tasks

Each personality manager provides a personality-specific set of application interfaces. This includes a method by which services are trapped (*i.e.,* system service transfer vector or system call trap) as well as an implementation for the trapped services. These interfaces are contained in a personality library that resides in the application task's address space and either implements the application interfaces locally or makes requests on the personality manager and other servers in the system.

Each personality library must be able to coexist (and possibly cooperate) with other personality libraries that might be associated with an application. Additionally, other servers in the system must define their services in a manner that supports multi-personality applications. For example, the I/O server should provide services so that an application could open a file with a VMS personality function and then read it using a BSD or POSIX [8] personality function. This leads us to view the application task as primarily consisting of three entities: the application image, support code inserted by each personality manager associated with the application (the personality libraries), and support code inserted by various servers used by the personality libraries to provide services to the application (the server support libraries). We envision the server support libraries as a form of proxy-code [9] that enables actions to be performed by the the client on behalf of the server. A good example of this is a file cache that reduces the number of I/O server requests by buffering data in the client's address space. Figure 3 illustrates the personality-related components of the application task and the servers associated with these components.

Personality managers process exceptions that occur during the execution of applications. Exceptions and software conditions that cause application termination return an error status to the application manager, and thence to the command interpreter. This allows exceptions to be processed within the application task, although their handling is subject to each personality library. This approach differs from VMS-on-Mach, where exceptions are delivered to the associated process server and then reflected back to the user library for processing.

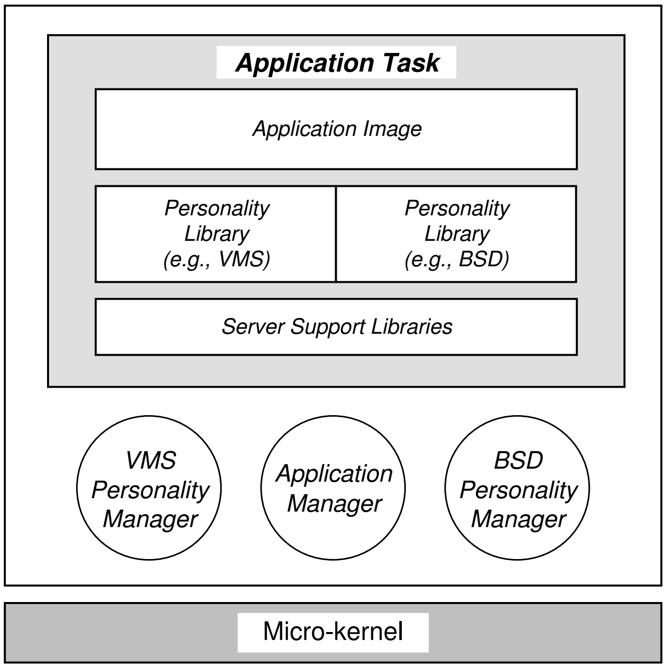

Figure 3 Application Environment

Generic Servers

A set of generic servers provide personality-neutral services with which personality managers and their associated personality libraries implement specific personality environments for applications. This includes file system servers, I/O servers, authorization and security servers, and synchronization servers. The underlying concepts of these servers are:

- Each server implements only a single set of related services. That is, servers are not overloaded with multiple unrelated services.
- Servers are designed for ease of testing. That is, a server is designed to function as an independent entity and can be tested on its own.
- Servers provide generic personality-neutral services. For example, file system servers should interact with personality-neutral file specifications.

4 Intrepid Workgroup Model

The Intrepid system model partitions personality environments and their underlying services into a set of independent servers that are used to provide a cohesive environment to the user. The Intrepid workgroup model combines several Intrepid systems into a distributed system that shares the collective resources of the individual Intrepid systems with members of the workgroup.

Figure 4 illustrates two workgroups within a heterogeneous network of independent systems. Each of these workgroups consists of several Intrepid systems that share resources and services. Resource sharing, however, is not confined to the workgroup, as each workgroup may provide resources and/or services to other systems and workgroups within the network.

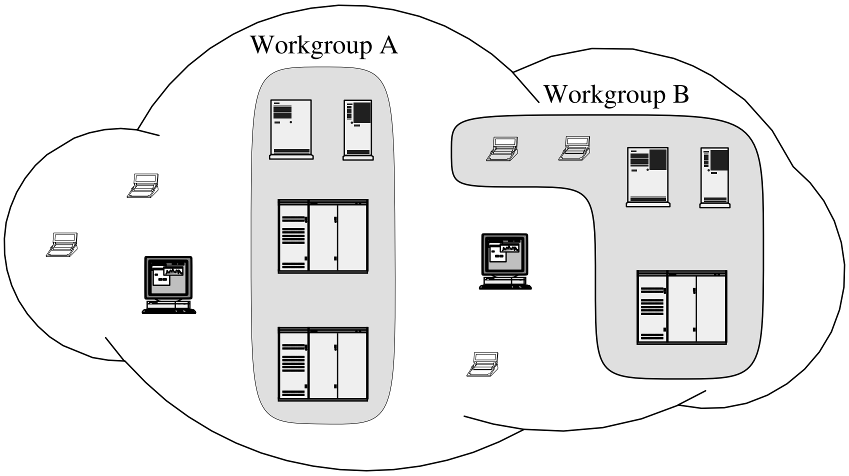

Figure 4 Intrepid Workgroup Model

We now define a workgroup manager, present on each system within the workgroup, that manages access to the workgroup's resources and services. Each workgroup manager is responsible for interacting with other systems in the workgroup to establish communication channels (via one or more network servers) used to provide access to the resources and services of the workgroup.

The workgroup manager allows clients to transparently utilize services provided by servers on other systems within the workgroup. As a result, certain servers are not required to be present on all systems within the workgroup. For example, the workgroup may have only a single authorization server. Similarly, a workgroup may have MS-DOS personality managers present only on 80486-class systems.

Peer-to-peer communication initiated via the workgroup manager is used to access services provided by different systems within the workgroup. For example, access to any device within the workgroup is requested from the local I/O server. The local I/O server, realizing the request is better serviced by (or can only be serviced by) a different I/O server within the workgroup, uses the workgroup manager as an intermediary to establish a communications channel with the remote I/O server. This channel to the remote I/O server is passed back to the client, and used for all subsequent access to the device.

The Intrepid workgroup model integrates a loosely-coupled network of Intrepid systems into a distributed system that shares the resources of its members. This loose coupling allows systems to be added and removed dynamically from the workgroup while maintaining a cohesive set of services to users and their applications. In addition, redundant servers may be >present within the workgroup, thereby encouraging the use of fail-over techniques to improve application availability.

Consider, for example, the workgroup shown in Figure 5. This workgroup contains a network of three systems: an Alpha system, a DECstation, and an Intel 80486 system. A DCL command interpreter and VMS personality manager run on the Alpha system, a *csh* command interpreter and BSD personality manager run on all of the platforms, and a DCL command interpreter and DOS personality manager run on the Intel 80486. An application manager, workgroup manager, and I/O server are present on all systems within the workgroup. Because of the interoperability and the distributive nature of Intrepid, a user can run their favorite 80486 BSD program from their DCL session running on the Alpha system. The application manager on the Alpha system requests the application manager on the 80486 (by means of the workgroup manager) to activate the requested application. An image server on the 80486 communicates with the I/O server running on the DECstation (which is acting as a disk farm) to load the BSD application. The BSD application runs on the 80486 with support from the BSD personality manager with all I/O directed back to the Alpha system. Likewise, a *csh* user logged into the 80486 can run their favorite Alpha-specific VMS program. The VMS application runs on the Alpha system using the VMS personality manager with I/O directed back to the *csh* command interpreter on the 80486.

This can be extended even further by including DCL and VMS servers on the DECstation and 80486 machines. This would allow users to develop new VMS applications under the DCL/VMS environments using other architectures. Existing Alpha-specific VMS programs can still be run under the control of the VMS personality manager server running on an Alpha system within the workgroup.

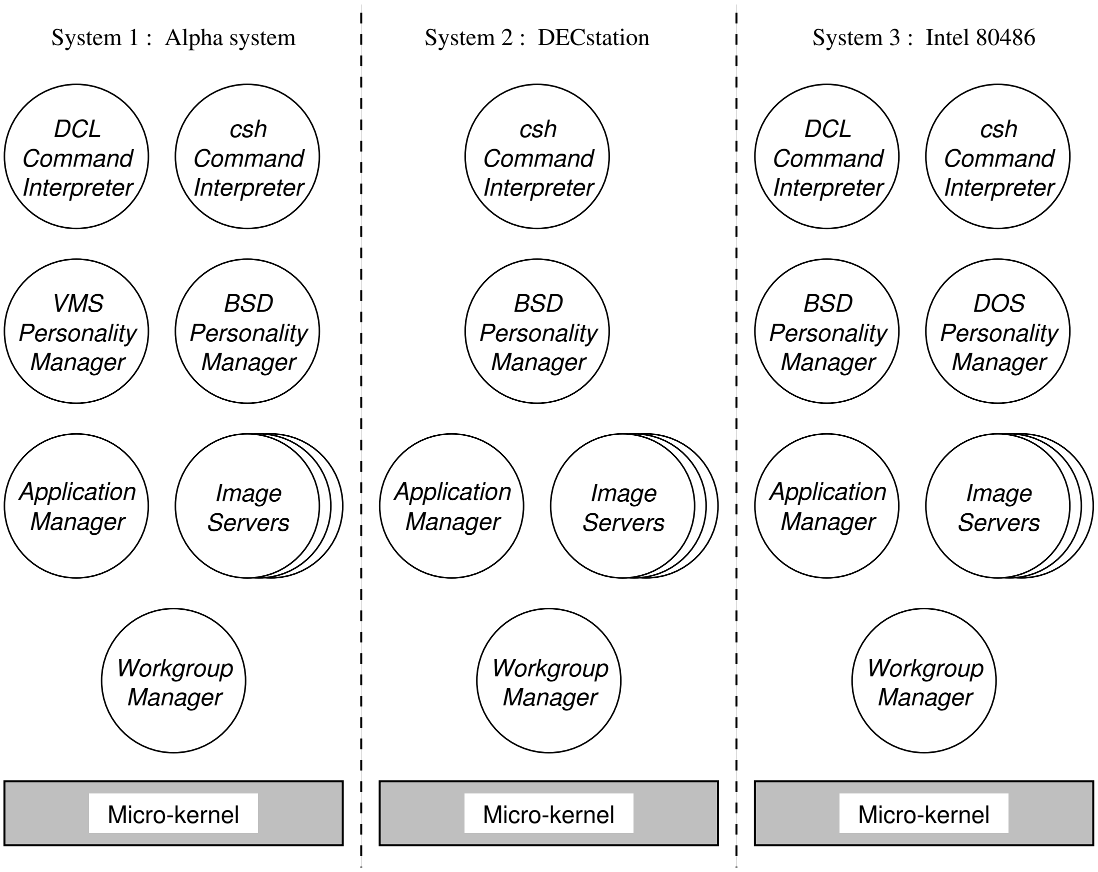

Figure 5 Intrepid Workgroup Example

5 Prototype Implementation

The VMS-on-Mach prototype implemented small portions of several servers including the executive server, the session manager (which VMS-on-Mach called the process server), a file-mapped memory server, an I/O server, and a logical name server. These servers were implemented as UNIX programs running on the 4.3 BSD-compatible environment provided with the Mach 3.0 micro-kernel. VMS-on-Mach servers existed alongside the BSD server and provided a subset of the VMS environment. This included DCL, VMS user images, and a handful of system services.

We decided to augment this prototype to support the Intrepid model. Since we had redesigned only the process server, all other aspects of the existing prototype could remain essentially intact. This meant that only the process server code needed to be reworked into a session manager, application manager, DCL command interpreter, and a VMS personality manager.

Our goal for the Intrepid prototype was to demonstrate that multiple servers could be used to provide separate pieces of various environments. Consequently, we chose not to address workgroups, networks, or the intermixing of services from multiple personality managers.

The VMS-on-Mach process server was true to the model in the partitioning of its implementation. Because of this separation, both in data structures and source files, we were able to rework the process server into its constituent servers with minimal effort. Most of the original process server was designed with the assumption that the code was single-threaded and that the data referred to a single process. Most of the effort in reworking the process server was in making the code thread-safe and in modifying the data structures to support multiple sets of data.

In VMS-on-Mach, to minimize implementation effort, we chose to load VMS images into the process server's address space and used Mach memory inheritance to pass the application image and user library code to the user space task. Additionally, several data structures were created in the process server and either inherited as read-only, or shared in the user space task. In Intrepid, the application manager creates the application task and uses the image servers and personality managers to load the application images and the environments they require. This meant that the separation of image format from environment needed to be clean, and that shared memory and inheritance were no longer suitable implementation mechanisms.

We defined a single set of services that are provided by all image servers. These services include load and unload operations and use data structures that are independent of image-file format. These services and structures provide only the basic services that are necessary to support VMS .EXE and UNIX a.out image-file formats. Further research is required to define a more complete set of services and data structures.

Loading a personality library into the application task's address space was neither a complex nor difficult task. We made use a Mach service to write data into the application task's virtual memory. We eliminated the shared memory regions and either passed the required information to the personality library's initialization routine or required the personality library to fetch the information (via RPCs) as needed. This required some additional processing during personality library initialization.

The application program's stack caused us the most problems. In VMS-on-Mach, this stack was doubly mapped into the user space task and the process server. We chose not to doubly map the stack in Intrepid as the proliferation of stacks from multiple applications could possibly overload the virtual memory of a personality manager. We observed that less than one page of stack is needed to create the application's initial call frames. Consequently, we build the application's initial call frames in a session manager private buffer that is written to the appropriate stack within the application task's address space.

Exception processing was also quite difficult. When an unhandled exception is reported, the application's call frames are traversed and a listing is made of all known call frames, return addresses, and stack-based arguments. As exception reports are not, hopefully, a common occurrence, we chose to handle them as a special case. When an exception report is generated, the active portion of the victim's stack is read into memory local to the personality manager. The exception report is generated from this local copy of the application's stack.

The Intrepid prototype, at present, contains the following new servers: a session manager, an application manager, a VMS personality manager, a DCL command interpreter, a VMSEXE image server, and an AOUT image server. To date we have successfully executed unmodified VMS images. We have also created a.out image files under UNIX that exclusively use VMS system services. These programs, which require the VMS personality environment to execute, have been successfully run by Intrepid. We are currently investigating building a *csh* command interpreter and BSD personality manager.

6 Performance

The process model of Intrepid is quite different from that used in VMS-on-Mach. In order to gauge the effectiveness of our Intrepid prototype, we ran several benchmarks and compared the results to data obtained from our prototype of VMS-on-Mach. Since the process was the only aspect of our model that significantly changed, we concentrated our measurements on process creation, image activation, process query, and exception processing. The following paragraphs describe the performance tests we ran and the results we observed.

To create a process in Intrepid, a thread and context area are created by the session manager. Once the user associated with the process is validated, a thread and context area are created by the selected command interpreter. In VMS-on-Mach, a new process server is started and the command language tables are loaded for each process. By multi-threading the session manager and using multiple servers to partition the work, process creation time in our prototype of Intrepid decreased by 67% to 0.1 seconds. Note that this time includes four RPCs to the executive server to issue time-stamped debug messages.

In VMS-on-Mach, the user library is statically linked into the process server whereas user images, the user stack, and the user environment are dynamically loaded into the process server's address space; user space tasks are created inheriting the process server's address space. Since multiple servers participate in the creation of the application task within Intrepid, memory inheritance cannot be used as an optimization. The personality library is built as a separate image that is loaded by the personality manager. After the application task is created by the application manager, image servers use the file-mapped memory server to activate the selected program into the application task's address space. The personality library is written into the application task by the personality manager using a Mach function. Our initial test, which measured the time required to load and run an empty image, shows that Intrepid performs 11% slower than VMS-on-Mach. Our second test, which is more characteristic of VMS programs, measured the time needed to run an application containing 500 pages of code and data of which 50% these pages were referenced. This test shows that Intrepid performs 1.5% slower than VMS-on-Mach.

In VMS-on-Mach, the executive server fields cross-process queries for process information, and redirects requests to the appropriate process server. Hence, requests from user images about the current process require a single RPC to the associated process server whereas cross-process queries require two RPCs. In Intrepid, there is only one session manager for the entire system, thereby reducing the number of RPCs necessary for cross-process queries by one. In Intrepid, however, not all information is available within the session manager, thereby requiring additional RPCs to gather the requested data. Our measurements indicate a comparable time for cross-process queries that require information from both the session manager and command interpreter, and a 75% increase for same-process queries that require information from both the session manager and the command interpreter.

In VMS-on-Mach, the process server receives user image exceptions and issues an IPC to the user library to initiate exception processing and handling. In Intrepid, exceptions are processed locally by the personality libraries within the application task. To measure the effect of this change, we ran tests that exercised both hardware- and software-generated exceptions. Our first test, which causes a hardware exception after establishing several nested VMS condition handlers [5], shows that Intrepid processes hardware-generated exceptions 28% faster than VMS-on-Mach. Our second test, which generates a software exception by using the VMS LIB$SIGNAL service [4], shows no significant change from VMS-on-Mach, which already processes these exceptions entirely within the user library.

Our initial measurements show that Intrepid process creation is significantly faster than VMS-on-Mach, whereas image activation is minimally slower for large programs. Process queries, however, are generally faster in VMS-on-Mach, while exception processing within Intrepid benefits from the localized processing within personality libraries.

7 Conclusions

In this paper we present an alternative view of a development environment: a user session interacts with multiple command interpreters to process commands, and with multiple image servers and personality managers to invoke applications. By separating the command language from the personality environment, we allow application developers to use any command language and combination of tools and personality environments to create applications. Applications can be viewed as black boxes, and users can now choose their favorite command language or even dynamically switch between command languages. Similarly, application developers can choose whatever combination of environments is best suited for their application. When a user runs the application, the correct environment is used regardless of the user's command language.

Besides taking interoperability to an extreme, this design presents a whole new definition of the process and operating environments. Consider again the example: `directory -size -date | more`. In a traditional *csh* implementation, the *csh* would execute in one process and new processes would be created to execute each of the programs specified in the command. In Intrepid, the *csh* executes on behalf of the user's session and requests the application manager to invoke both applications simultaneously. The application manager requests assistance from the VMS and BSD personality managers to support execution of the requested applications (with I/O between the applications already set up by the command interpreter). Similarly, the DCL `spawn` command need not create a new session to invoke a command file or run an application. Besides simplifying management of the system and reducing the general overhead associated with each session, user response time is improved.

By separating and isolating the command language, we open the door to new opportunities. Revisions to existing command languages can be created as separate command interpreters providing complete backward compatibility. Windowing interfaces (*e.g.,* OSF/Motif[1] and Open Look) can be implemented as command interpreters that are not tied to any specific personality environment. New command languages (*e.g.*, voice input systems) can be readily incorporated as command interpreters. A new capability, made available for one command language or programming interface, can be immediately used by any user or application program, and can also be made available to other command languages or programming interfaces with relatively minor incremental effort.

The separation of personality environments allows existing and emerging interfaces and standards to be easily added and made available to applications. For example, by providing UNIX, VMS, and POSIX personality environments, applications can intermix POSIX services with VMS and UNIX services. Additionally, new interfaces (*e.g.*, 64-bit interfaces) can be implemented as separate personality environments and used in conjunction with existing interfaces supplied by other personality environments.

Our previous efforts led us to conclude that VMS can be layered on top of a micro-kernel and partitioned into multiple servers that can be shared by multiple environments. This new work leads us to conclude that by partitioning the system into servers with basic functions, VMS and other environments can make use these servers to achieve new levels of interoperability. In addition, new command languages can be easily added as can new application interfaces.

8 Future Work

This work investigated the environmental aspects of interoperability. We envision even greater interoperability as one explores networking and clustering of these environments specifically with regard to I/O and file systems. Our prototype of Intrepid achieved file system interoperability by supporting only a single file system. We envision, however, a more hierarchical system where multiple on-disk formats are supported by multiple file system servers.

This paper outlines an environment in which command languages can use other command languages. Additional research is needed to determine exactly how this interoperation should be expressed. For example, should a user be able to type `ls -l` at a DCL prompt and expect a directory listing? One possibility is that interoperation in specified in native syntax. In this example, the user would need to define a DCL foreign symbol (similar to a *csh* alias) for `ls` that references a specific image. Assuming the application manager (or file system) can determine that the image is a BSD image, then typing `ls -l` would do the same thing that it does at the *csh* prompt. However, typing `ls -l *.a` presents a more complicated problem, as it may require interaction with the *csh* command interpreter. Such efforts can now be very successful since application compatibility is no longer an issue.

The actual semantics of multiple personality managers requires additional work. Ideally, the application manager or personality managers should be able to detect personality manager switches. In some cases, where trap mechanisms are different, detection might be trivial (or even not needed). If binary compatibility is not an issue, a unified mechanism might be a viable approach.

[1]OSF/Motif is a registered trademark of Open Software Foundation, Inc.

9 References

[1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation For UNIX Development", *Usenix Association Proceedings,* Summer, 1986.

[2] Digital Equipment Corporation, *VMS Linker Utility Manual,* Order No. AA-LA62A-TE, April, 1988.

[3] Digital Equipment Corporation, *VMS DCL Concepts Manual,* Order No. AA-LA10A-TE, April, 1988.

[4] Digital Equipment Corporation, *VMS Run-Time Library Routines Reference Manual,* Order No. AA-LA76A-TE, April, 1988.

[5] Digital Equipment Corporation, *VMS System Services Reference Manual,* Order No. AA-LA69A-TE, April, 1988.

[6] M. Gien, "Micro-kernel Design", *UNIX REVIEW,* vol. 8, no. 11, pp. 58-63, November, 1990.

[7] R. Goldenberg, and L. Kenah, *VAX/VMS Internals and Data Structures, Version 5.2.,* Digital Equipment Corporation, Order No. EY-C171E-DP, 1991.

[8] *IEEE Standard Portable Operating System Interfaces for Computer Environments, IEEE Std. 1003.1-1988,* Institute for Electrical and Electronics Engineers, Inc., 1988.

[9] K. Loepere, editor, *Server Writer's Guide,* Open Software Foundation and Carnegie Mellon University, 1990.

[10] R. Rashid, "A Catalyst for Open Systems", *Datamation*, vol. 35, pp. 32-33, May 15, 1989.

[11] UNIX Programmer's Manual Reference Guide, 4.3BSD, April, 1986.

[12] C. Wiecek, C. Kaler, S. Fiorelli, W. Davenport, and R. Chen, "A Model and Prototype of VMS Using the Mach 3.0 Kernel", *USENIX Association Proceedings,* April, 1992.

[13] D. Scherrer, and V. Vaughan, *UNIX REVIEW*, vol. 10, no. 4, pp. 32-38, April, 1992.