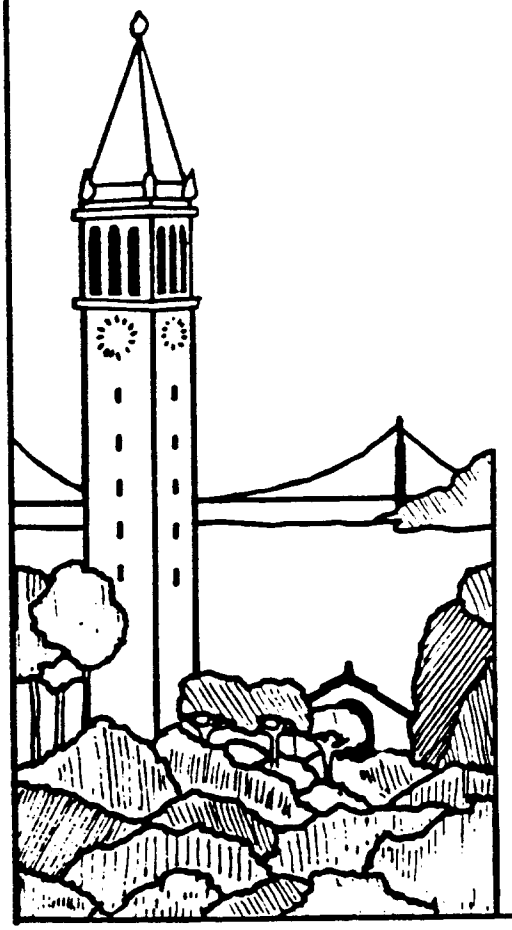
The DASH Project:

Issues in the Design of

Very Large Distributed Systems

David P. Anderson
Domenico Ferrari
P. Venkat Rangan
Shin-Yuan Tzou



Report No. UCB/CSD 87/338

January 1987
PROGRES Report No. 86.10

Computer Science Division (EECS)
University of California
Berkeley, California 94720

The DASH Project: Issues in the Design of Very Large Distributed Systems

David P. Anderson
Domenico Ferrari
P. Venkat Rangan
Shin-Yuan Tzou

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, California 94720

January 1, 1987

ABSTRACT

A *very large distributed system* (VLDS) is one based on a fast wide-area network connecting numerous organizations and individuals. The design of a VLDS involves problems and issues not present for smaller systems. These issues are centered in the areas of naming, communication paradigms and architectures, security, and kernel architecture.

DASH is a research project aimed at investigating VLDS design issues. The goals of the DASH project are 1) to predict the advances in computer and network hardware and application software that apply to VLDS; 2) to propose a set of design principles for VLDS, and 3) to experimentally validate these principles by building and testing a prototype VLDS kernel.

This report is a high-level view of the DASH project as of December 1986. After summarizing our ideas about the potential uses of VLDS's and our assumptions about the environment in which they will exist, we examine several design areas. In each area, we offer some possibly controversial assertions, attempt to justify these assertions, and describe how the assertions have guided the design of DASH.

This research was supported by the Defense Advanced Research Projects Agency, by the IBM Corporation, by Olivetti Sp.A., by MICOM Interlan, Inc., and by the University of California under the MICRO program. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or of the US Government.

1. INTRODUCTION

The DASH project is concerned with the design of *very large distributed systems* (VLDS). By this term, we mean that:

- The system is large in the following senses: *numerical* (it contains thousands or millions of connected hosts), *geographical* (hosts may be thousands of miles apart), and *administrative* (the system encompasses hosts and networks owned by many organizations and individuals).
- The system offers access to non-local resources such as databases, processing power, software, and communication with remote human users.
- This access may be *transparent* in the senses that 1) at some level (perhaps the user interface) the same syntax can be used to access both local and remote resources, and 2) the performance difference between local and remote access is not excessive.

Such a system is preferable to a collection of unintegrated local distributed systems for several reasons: the foremost is that it allows efficient resource sharing on a much larger scale.

Many of the assumptions underlying the designs of current small distributed systems do not hold for VLDS. Fundamental differences exist in the following areas:

- Security
- Naming
- Communication paradigms and architectures
- Kernel architecture

The DASH project is studying each of these areas in the context of VLDS. We have considered tradeoffs in the design space, and have suggested various novel approaches. We are in the process of developing a prototype kernel for a VLDS that can be used to test our assumptions. However, our immediate goal is to investigate principles of VLDS design rather than to build a usable system.

Previous projects have considered VLDS problems such as scalability of nameservers [37], file services with many clients [36] and distributed text retrieval [18]. A large-scale distributed UNIX system is described in [47]. These projects, for the most part, address restricted problems and develop solutions based on technology that will soon be outdated. DASH, on the other hand, takes a unified approach and seeks solutions that will not be made obsolete by foreseeable technology advances.

This report summarizes the thinking behind the DASH project. It consists of

- A projection of changes in hardware and software technology that apply to VLDS (section 2).
- A set of principles, goals, and non-goals for VLDS design (section 3).
- A discussion of the following aspects of distributed system design:

  General structure (section 4)
  Naming and authentication (section 5)
  Network communication (section 6)
  The "service" abstraction (section 7)
  Execution environments (section 8)
  Kernel structure (section 9)
  Software engineering (section 10)

In each area we state our current design decisions (printed in boldface). Each decision is followed by a rationale based on the principles and technology forecast. Each section concludes with a description of the current DASH design in that area.

There are many technical problems in VLDS design beyond those discussed here. For example, the problem of accounting is important and difficult, particularly if network traffic is not free. In addition, there are obstacles (such as persuading a critical mass of people to agree on a design) that are political or pragmatical rather than technical.

2. TECHNOLOGY PROJECTIONS

The design of an operating system is heavily influenced by assumptions about its hardware, software, and administrative environment. What should these assumptions be for a VLDS? Our contention is that they should be as long-range as possible, because 1) once a VLDS is adopted, it will be difficult to modify or replace; 2) there are many potential uses of a VLDS that will become feasible only with technology that is a few years away, but that must be accommodated in its basic design.

2.1. Hardware Forecast

Wide-area networks based on fiber optics will provide end-to-end bandwidth comparable to that of current LAN's (10 Mb/s) and eventually greater than the memory bandwidth of current hosts (100 Mb/s to several Gb/s). Improved switching technology will yield low average latency: perhaps 10-20 milliseconds for switching, plus signal propagation time (about 30 milliseconds for 3000 miles). Furthermore, the network will provide optional *guaranteed-performance soft virtual circuits*: in return for a client promise of bounded outstanding data over a point-to-point logical channel, the network will guarantee a minimum available bandwidth or bounded delay on that channel. The feasibility of such a network is suggested by the results in [48], [40] and [27].

As VLSI design and fabrication technologies advance, processor and memory chips will continue to decline in price and increase in capacity. Low-priced workstations may soon have a 20 MIPS processor and 100 MB of main memory [2]. Devices capable of performing single-key encryption/decryption at 10-20 Mb/s will become available [23]. Special-purpose processors for tasks such as logic programming and signal processing will become more prevalent.

Peripheral technology such as high-resolution color graphics, mass storage, and digital audio devices will continue to increase in performance but will not fall in price as rapidly as IC's.

A trend towards multiprocessors is suggested by these assumptions. The economic impetus for this type of architecture is that the expensive parts of a system (displays, permanent storage, I/O systems) can be best used by placing as much (cheap) processing power close by, e.g. on the same memory bus [15]. In particular, shared-memory multiprocessors will become common.

2.2. Software Forecast

The demand for CPU cycles (in, for example, graphics and artificial intelligence applications) exceeds projected increases in processor speed. We contend that this demand can often be satisfied more economically by using network parallelism (i.e. by executing processes concurrently on different hosts) than by using specialized hardware [14], general-purpose parallel hardware [22] or supercomputers.

The set of all processors on a VLDS can be viewed as a *processor bank* [13] numbering perhaps in the millions of processors. This presents the opportunity both for load-balancing with sequential program execution, and parallel computation at many granularity levels. It also introduces technical problems relating to 1) managing and using load information for large numbers of processors, 2) efficiently dispersing and collecting the results of remote computations, and 3) protection and security.

The sophistication and diversity of user interfaces will continue to increase. A distant resource such as a news service might offer a graphics/audio based interface with the same bandwidth requirements as a local mouse-based editor. Because long-distance network delays are inherently high, such services will use techniques such as data streaming, read-ahead, and local caching to achieve the needed performance.

Many communication applications will be possible in a VLDS:

- Commercial applications such as advertising, sales, and remote banking.
- Interpersonal communication forms such as mail, telephone, facsimile, and video conferencing.
- Distribution of digital audio and video entertainment and news.

All the proposed applications of Integrated Services Digital Networks (ISDN) [16] can be supported (and improved upon) by a VLDS. The value of these services will be greatly enhanced if the VLDS is as widespread and pervasive as the current telephone system.

2.3. Administrative Forecast

Because of falling hardware prices, ownership and control of hosts will continue to shift from central (corporate or departmental) to distributed (individuals or small groups). As a result, physical security of hosts cannot be assumed in general, although specific security assumptions could exist. For example, a workstation might be accessible only to its owner; a group might assume that a particular host is physically accessible only to trusted administrators, or only to members of the same group. Similarly, the physical security of networks cannot be assumed, although the security of a particular subnetwork may be assumed by a particular individual or group.

As more hosts are owned by people who are not system experts, administrative problems (such as configuration of kernels and distribution of software releases) will increase. Companies will offer this type of administration as a product; many security-related problems will arise from this.

3. PRINCIPLES, GOALS AND NON-GOALS

A VLDS must serve over a long period of time and must accommodate many types of usage. Hence it has the following conflicting requirements:

- To provide enough *flexibility* and *performance* to allow customization, research, and evolution at both the application and the system levels.
- To provide enough *structure* to encourage internal compatibility and to eliminate the need to reinvent basic functionality.

What is needed is a "software chassis" in the style of the V system [9] but for VLDS's rather than LAN-based systems. This leads to the following VLDS principles (see [25] for a discussion of general principles of system design, many of which apply to VLDS).

> **Principle of Minimal assumption: a VLDS should not impose language or protection models, and should avoid restrictive hardware or usage assumptions.**

Rationale: with the existing diversity of languages and applications, there are few universally acceptable models or assumptions. The power of a VLDS depends directly on how widely it is used, and the more assumptions it makes, the less it will be used.

> **Principle of Performance: the basic facilities of a VLDS should not sacrifice speed for a high level of abstraction.**

Rationale: a VLDS must support diverse and unpredictable abstractions. It may not be possible to implement new abstractions efficiently on top of incompatible abstractions.

Principle of Location Independence: the mechanisms for (and the possibility of) performing remote operations should not depend on the locations of the entities involved.

Rationale: this is a practical necessity for large-scale distributed computation, since it frees programmers from site-dependent concerns. It also means that the host to which a user is logged on (which might vary considerably, e.g. during travel) need not affect the interface that he sees. This principle implies that protection should be provided by location-independent mechanisms rather than network "firewalls".

"Clustering" in distributed systems is motivated by the assumptions that long-distance communication is inherently slow and insecure. We make opposite assumptions, as described in section 2.1. However, there do exist locality effects (such as the performance and security characteristics of local-area networks) that should be recognized and exploited in a VLDS.

Principle of Autonomy: mandatory dependencies and trust requirements between hosts and subsystems should be minimized.

Rationale: mutually mistrustful organizations inevitably demand that their subsystems function independently; i.e. that they not require outside services, and that they not be required to provide any services to the outside. When they do interact, subsystems must be free to select their own policies controlling the interaction. Hence the operation of a VLDS cannot depend on a *central authority* such as a central naming or authentication service. Another consequence is that a notion of *ownership* must exist for many components of the system.

Principle of Scaling: the average speed of remote resource access should not depend on the number of hosts.

Rationale: the power of the system for parallel computation increases with its size. From the viewpoint of human communication applications, the total utility of a VLDS grows more than linearly with its size, since a system with n users provides n^2 logical communication paths. Therefore, a VLDS should be designed to perform well with millions or many millions of hosts world-wide. In particular, broadcast communication will be prohibitively expensive and should not be relied on in the system design.

3.1. Non-Goals

To be effective as a software chassis, a VLDS must be kept simple. Hence non-essential "features" should be left out. Specific examples will appear in later sections; in general:

Compatibility or interoperability with any existing system should not be pursued.

Rationale: the abstractions offered by existing centralized systems, such as the UNIX model of process creation and control, are inappropriate for VLDS. Interoperability with existing distributed systems would add compromises, complexities, and performance degradations to a VLDS. A well-designed VLDS, restricted to a LAN, can offer the same functionality and performance as a LAN-specific system. Hence there is no reason to retain such systems.

Compatibility with a widely-used system would make existing software available. However, we contend that many programs (such as compilers and text formatters) use primarily the file system interface and will be easy to port to a VLDS if it offers an appropriate file service. Programs that are difficult to port (command interpreters and programs that use IPC) will become unnecessary or will be replaced by more useful programs.

4. GENERAL STRUCTURE

What should be the general structure of a VLDS? To discuss this, we need a general framework for describing distributed systems. We use the following decomposition:

- Network communication structure

  The system is seen as a hierarchy of network protocols in the style of the ISO 7-layer model [20]. Some slots may be occupied by fixed protocols (perhaps naming and data transport) while others are open (application- or service-specific protocols). Hosts must obey these protocols, but are otherwise viewed as black boxes.

- Execution environment

  At this level, hosts are seen as sites of process execution. A host provides an *execution environment* consisting of a mechanism for initializing processes, and a semantics for program execution (i.e. an instruction set extended by system calls). A UNIX-style execution environment might load programs from disk files, based on the protocol of a particular type of file service. In contrast, a capability-based environment might initialize processes from process descriptors containing capabilities to memory segment objects.

- Implementation structure

  This is a logical structure for a *kernel* supporting the network communication structure and providing an execution environment. It is embodied in the machine-independent part of the kernel's source code.

- Implementation

  This is the kernel on a particular host or set of similar hosts.

Many distributed systems blur the distinctions between these levels; the assumption of homogeneity makes this easy to do. It is important to make these distinctions in the design of a VLDS, since there may exist many execution environments, implementation structures, and implementations (see sections 8, 9 and 10).

4.1. Network Communication Structure

We now concentrate on the first of the above components, network communication structure. This level can be further subdivided into layers, each with its own naming system and protocols:

(1) Low levels such as physical and data-link communication. Names are hardware network addresses.

(2) Multiplexed data transport. Names are logical network addresses and port numbers.

(3) The *client/server model*, in which communicating processes have asymmetric roles. Symbolic naming and protection mechanisms may be present.

(4) The *object model*, that extends the client/server model by adding some combination of protection, typing, language support, atomicity, and communication paradigm.

Existing distributed systems vary in the levels to which they allow access. Eden [4] provides access only at the object invocation level. The V system [9] provides an interface that is intermediate between (2) and (3) in that it leaves protection and symbolic naming up to the servers. Sun UNIX supports the client/server model with its RPC system [28], and also exposes the level (2) interfaces (UNIX sockets).

In distributed systems such as Locus [49] and Sprite [50], directly-accessible services are kernel-provided and are restricted to a fixed set of service types. Others, such as V, Eden and Mach [1] support *user-defined services* by providing IPC and naming facilities.

A VLDS should support a client/server model with user-definable services, and should also make transport-level communication accessible.

Rationale: as discussed in section 7, a VLDS must provide flexible service support; it is not enough, for example, to provide only a shared file service. Basic client/server functionalities such as naming and standardized access protocols are needed to encourage development of globally-accessible services, and need to be placed in the kernel for efficiency. Some applications must be able to bypass these mechanisms; for example, a large-scale distributed programming system may need to arrange communication between unnamed processes. Hence access to lower-level communication is needed.

4.2. The General Structure of DASH

The DASH network communication structure consists of a novel transport system (described in section 6) that is directly accessible and is also used as the basis for a generalized client/server model (section 7). Services provide their own internal naming and authorization, based on a global naming system that is used for other purposes as well (section 5).

DASH provides a framework for multiple execution environments, but provides a canonical execution environment (described in section 8). This environment is based on a local message-passing system that is used for network communication, local communication, process control, exception handling, and system calls.

5. NAMING AND PROTECTION

5.1. Scope and Permanence of Naming

The *nameable entities* in a distributed system may include hosts, users, services, objects (logical entities within a service), processes, communication endpoints, protocols, messages, words of memory, and so on. Nameable entities can be roughly classified as *transient* or *permanent* according to how often their mappings change; for example, hosts, users and services are permanent relative to objects and processes.

Similarly, names can be classified as *global* and *local*. An entity with a global name could exist anywhere; the location of an entity with a local name is restricted, usually to the local host.

Within the global name space, we distinguish between *system-level* names (managed by a special naming service with hardwired protocols) and *service-level* names (provided by an arbitrary service to name internal objects or specializations).

System-level global naming should name only permanent entities, and should provide only eventual consistency on updates.

Rationale: the principles of Performance and Minimal Assumption suggest that as few entities as possible have system-level names. For example, the system should not name objects within services since some services might use special naming schemes.

Eventual consistency considerably simplifies nameserver design ([44], [37]) and is all that is needed for many purposes. Applications that have other requirements on naming semantics (such as atomicity of updates) can use specialized naming services. Similarly, temporary entities can be given global names by specialized services (such as a "rendezvous server" for processes).

5.2. Form of Naming

Some existing distributed systems use *capabilities* as a combined basis for both naming and protection. Examples include Amoeba [31], Mach [35], and Eden [4]. Typically, a symbolic naming (directory) service is built on top of the capability mechanism.

System-level naming of permanent entities should be symbolic, with name resolution (but not protection) done at the kernel level.

Rationale: there are several reasons to avoid basing a VLDS on capabilities:

- Name resolution is inherently less efficient for processes off-loaded to a distant host since directory servers would generally be near their owner. Also, name resolution must always be done in user-level servers.
- It imposes a protection paradigm, contrary to the principle of Minimal Assumption.
- Many permanent resources will be accessible in some way to the entire world. A capability-based system uses one directory entry per (user, accessible resource) pair, requiring enormous amounts of storage. Furthermore, widespread distribution and storage of capabilities would weaken security.

Capability-like mechanisms are, however, well-suited to naming temporary objects. In DASH, such capabilities are be associated with a temporary secure communication channel through which the object is accessed. Protection is exercised in the formation of these channels. This avoids the above problems.

System-level global naming should be source-location-independent: that is, a given name should refer to the same entity no matter where the reference is made.

Rationale: systems in which each host has a separate name space violate the principle of Location Independence. Schemes in which each process has its own name space (e.g. per-process UNIX mount tables) are potentially location-independent but involve significant overhead and complexity.

System-level global naming should be target-location-independent: that is, a name should not imply the location of an entity.

Rationale: the service abstraction (section 7) may demand this type of location-independence; for replicated services, there may be no notion of location to begin with. In the cases where the location of an entity is needed, it can be supplied by other means (such as information in the name service entry).

Given that system-level global naming is to be symbolic, it remains to decide the form of the name space. Some existing systems (such as V) use a flat name space; attribute-based naming has also been proposed [38].

The global name space should be hierarchical and tree-structured.

Rationale: it must be possible for an organization to have authority over a portion of the global name space. It must be able to further subdivide this into subspaces to be used by its autonomous suborganizations. There can be no a priori limit on the depth of this subdivision. The simplest and best-understood way to provide this facility is with a tree-structured name space. In addition, this structure allows localization of trust in name resolution.

It has been proposed [26] that name services provide various features such as context mechanisms ("current directory" and "search path"), aliasing (nicknames) and database queries ("wildcard" or regular-expression queries, for example).

Context mechanisms, aliasing and general lookup queries should not be included in system-level global naming.

Rationale: the principle of Minimal Assumption applies, since there is no consensus on these features. They can be provided at a higher level, if needed.

5.3. Authentication and Protection

Having decided that capability-based protection is not generally inappropriate in a VLDS, authentication based on user names is needed.

Authentication should be based on global symbolic user names, and should use public-key encryption with per-user key pairs.

Rationale: user names should be part of the global tree-structured name space for reasons given above. The use of public-key rather than secret-key encryption as the authentication mechanism is suggested because 1) keyservers can be replicated with no reduction in security, and 2) a trusted central authority is not needed.

5.4. Naming and Authentication in DASH

There are several types of naming in DASH.

5.4.1. Permanent Resource Names

The entities with system-level global names in DASH are:

- Owners: these might correspond to individuals or to "roles". Each owner name is mapped to a public key, and optional information such as the real-life name of person (or description of the role), the name of a "mail service" by which the owner can be reached, and a physical address.
- Hosts: each is mapped to 1) a set of network addresses for the host, 2) the name of the *permanent owner* (who is allowed to change this entry), 3) one or more names of *tenants*. Tenants are likely owners of the host's kernel, and their names can be used as hints in establishing a secure channel to the host (section 6). 4) the type of the kernel running on the host (section 5.4.2).
- Services: each is mapped to a set of host names and an owner name.

The global name space is tree-structured and is managed by a set of *name services*, each with authority for one internal node. The leaf nodes are the entities listed above; name services also act as public key servers for their owner names. The upper parts of the tree will correspond to organizational hierarchy, similarly to the Internet Domain naming system [30] (see figure 1).

Each name service entry has an owner, and can only be changed by that owner. Name services are, in general, publically readable.

The name services are accessed by a fixed protocol that supports both lookups and directory reading. However, they may use a variety of resolution algorithms and caching policies, depending on the security and consistency needs of their clients.

One consequence of global naming is that all entities have long names. This poses two problems:

- Naming is potentially cumbersome to users and programmers, since many components may be involved.
- Name resolution can have significant overhead [29]

Possible solutions to the first problem involve context mechanisms built into user interfaces and language libraries that automatically prepend an initial segment to some or all names. The second problem has been addressed in other systems by various caching schemes [29]. The DASH kernel prototype uses an approach in which user processes can obtain "handles" on pathnames whose resolutions are cached in the kernel. Names are then represented by a pair consisting of a handle on an initial component of the name, followed by the remainder of the name in symbolic form.

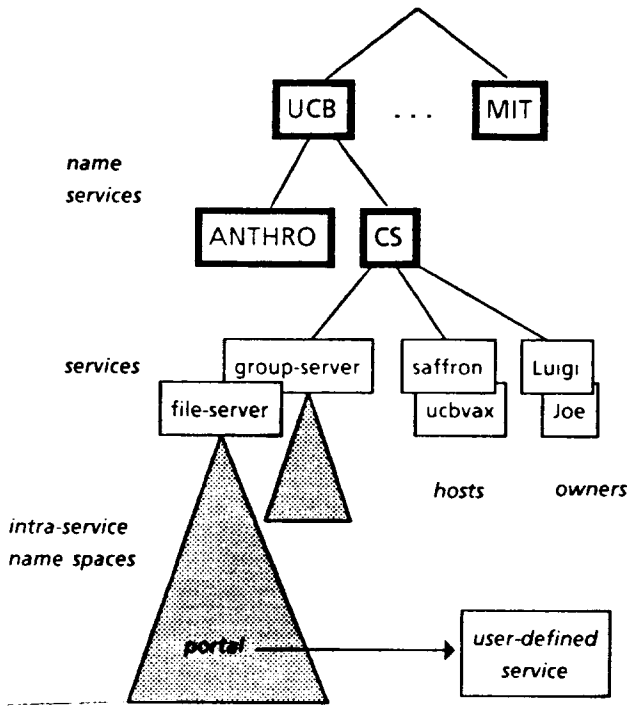


Figure 1: The DASH Global Name Space

5.4.2. Permanent Non-Resource Names

A second class of entities are global, but do not correspond to a resource and have no owner. These include:

- Data representation types (section 6.4).
- Execution abstractions (section 8).
- Machine architecture types.
- Stream protocols (section 6.4.1).
- Kernel types: each corresponds to a particular DASH kernel implementation, and is mapped to lists of the execution abstractions and stream protocols supported by that kernel.

The names of these entities are (unique ID, version number) pairs. The identifier part is assigned by a central authority, and distribution of these identifiers is done by ad-hoc means.

5.4.3. Temporary Names

Temporary entities such as processes, ports, and name handles are part of the execution environment, not the base system. In the DASH canonical execution environment,

user processes use kernel-supplied identifiers to refer to these entities. Other temporary entities, including all remote temporary entities, are accessed through secure communication channels; the port identifiers act as temporary capabilities to these entities.

As an example, there is no way of directly naming a random remote process or address space. Thus, entities that never are referred to entail no naming overhead. During process creation, however, it is possible to set up a communication channel that can be used to access or control the process. This might be used, for example, by a "process server" for a distributed programming facility, which might handle both load-balancing and random process naming and communication.

5.4.4. Intra-Service Names

The DASH service mechanism (section 7) allows services to extend the global name space below their own name. A file service, for example, manages the naming of its files however it chooses, hierarchically or otherwise. From the client's viewpoint, there is no distinction between system-level naming (used to specify the service) and intra-service naming. The kernel locates the service using an initial part of the name given by the client, and passes the remainder to the service without interpretation.

6. NETWORK COMMUNICATION ARCHITECTURE

Communication architectures can be divided into levels, as in section 4.1. In this section we discuss the middle levels (data transport and service access) in the context of VLDS. We are not concerned here with 1) the network level and below, 2) the highest levels (service-specific protocols), or 3) intramachine communication.

6.1. Network Transparency

Systems such as Mach [35] and Quicksilver [21] provide network communication that is a *transparent* extension of local communication in that it provides the same syntax and, to some extent, semantics. In both cases, this is done by interposing "network server" processes that use fixed (reliable) communication protocols. Quicksilver couples this communication with distributed transaction management, making remote operations semantically similar to local operations.

No attempt should be made to give network communication the same semantics as local communication.

Rationale: such a design forces all clients to pay for the semantics of local communication (sequencing and reliability) even if they do not need it. Network performance and reliability characteristics should not be hidden; they should be exposed so that users can write applications that work efficiently. The mechanisms for addressing and security in Mach rely on broadcast and a central authority, which are not feasible in a VLDS.

6.2. Facilities and Protocols

The *communication facility* offered by a distributed system is characterized by its synchronization and reliability properties. With respect to process synchronization, facilities can be loosely categorized as follows:

- *Request/reply*: communication is done during synchronized "message transactions" during which the client is blocked.
- *Session-oriented*: the start and end of a "session" are synchronized; the intervening messages may or may not be.
- *Unsynchronized.*

A facility is implemented using *data transport protocols*, that likewise can be categorized as *request/reply, stream-oriented,* or *datagram*. There is not a strict correspondence

between facilities and protocols; for example, Sun RPC can use TCP connections [28]. However, a facility can usually be implemented most efficiently using a protocol that has similar semantics.

It has been observed [41] that 1) there is a range of semantics for request/reply communication (such as various levels of reliability), and that 2) the different semantics admit implementations of varying efficiency, as measured by message exchanges, processing at each end, and the amount of state required at each end. Specialized request/reply protocols are discussed in [11], [7], [10] and [9].

Similarly, session protocols may provide a range of different semantics in terms of reliability, number of data streams, real-time performance, flow control, data boundaries, out-of-band data, and so on. For a particular semantics, there may be a range of possible protocols, among which the best choice depends on network characteristics.

Stream-oriented protocols, and a session-oriented communication facility, should be supported.

Rationale: there are several factors here.

- Pipelining of packets is needed to utilize the bandwidth of high-delay networks, and a session-oriented facility is needed to provide efficient access to stream protocols.
- The session model allows the cost of naming, authentication and authorization to be amortized over longer periods.
- Caching and network read-ahead, necessitated by high network delay, may be more efficient and easier to implement in the session model.
- Some applications will have real-time communication requirements such as minimum bandwidth or maximum delay. Network architectures that support such guarantees (section 2.1) will so on a per-session basis.

The stream-oriented protocol architecture should allow independent data channels per stream.

Rationale: an interface to a service may involve several different types of traffic, each of which has different semantic or performance requirements. For example, a service might use separate channels for voice data, graphics data, and control messages.

The stream-oriented protocol architecture should allow protocols to be flexibly configured (i.e. stacked, and bound to data channels).

Rationale: a particular semantics can often be realized by composing a set of simpler protocols [34]. A single protocol may inherently involve more than one data channel, or may be able to exploit dependencies between data channels (such as character echo in the Telnet protocol) for efficiency gains.

A request/reply facility supporting *maybe*, *exactly-once-type-1*, **and** *at-least-once* **semantics should be provided.**

Rationale: request/reply communication is needed for system-level activities, and each of the three reliability levels is appropriate for one or more such activities. For example, distribution of cacheable hints can use *maybe* semantics, name lookups and other idempotent operations can use *at-least-once* semantics, and process creation needs *exactly-once* semantics. It has been shown [9], [41] that special kernel-level protocols are can optimize each of these communication semantics.

All communication should pass through a "sub-transport" layer that handles network interface scheduling.

Rationale: many applications, particularly those involving voice, interactive interfaces, and real-time control, will have soft real-time constraints on communication. This might take the form of maximum request/reply delay, maximum delay on a stream, or minimum

bandwidth on a stream. For example, one channel of packet voice might require 64 Kb/s bandwidth and 40 ms maximum delay [45]. In hosts of the future, the major delay sources and bandwidth bottlenecks will probably be in the network interface and its driver software. Hence the only way to support the real-time guarantees is to schedule all network traffic using a non-FIFO discipline.

6.3. Network Security

The network security mechanism in a VLDS must provide a choice of security levels: none, authentication only, and authentication plus privacy. It must not interfere excessively with network performance, since the bulk of network traffic will require at least authentication.

The network security mechanism in a VLDS should be placed in the sub-transport layer.

Rationale: placing security in the sub-transport layer makes it possible to use single host-to-host secure channels. Multiple secure channels between separate user-level processes (often cited as being necessary for "end-to-end" security) are no more secure than host-to-host secure channels, and the latter are potentially more efficient for the following reasons [5]:

- There are fewer secure channel establishments and fewer secure channels at any point.
- There is the opportunity for *encryption piggybacking* (amortizing the cost of encrypting headers).
- There is less duplication of effort at higher levels.
- The need for three-way handshakes in upper-level protocols is eliminated.

The network security mechanism in a VLDS should use public-key encryption for initial authentication, and single-key encryption subsequently.

Rationale: as discussed in section 5, public-key encryption is the preferred means of initial authentication. However, only single-key encryption appears suitable for high data-rate implementation [23].

6.4. The DASH Communication Architecture

DASH IPC is divided into the following levels (see figure 2):

- The internetwork, network and lower layers are unspecified. The current implementation uses either the DARPA Internet Protocol or a specialized LAN protocol, depending on the destination.
- All network communication passes through a *sub-transport layer* that handles interface scheduling and stream multiplexing. This layer also implements an *authenticated datagram protocol* (ADP) that provides authentication and optional privacy. All network communication is in the form of *ADP messages*, that may contain several unrelated *ADP client messages* and whose transmission may involve multiple network- or internetwork-level packets.
- The sub-transport layer does multiplexing/demultiplexing of *streams*; this facility is used directly by stream-oriented protocols. All ADP client messages with a zero stream ID belong to the *DASH request/reply protocol* (RRP). RRP supports the three reliability levels of section 6.2.
- The *DASH kernel/kernel protocol* (KKP) is based on RRP and consists of a set of request/reply types. KKP is used for various purposes, such as service access (service location, session establishment, etc.; see section 7), and remote kernel access (section

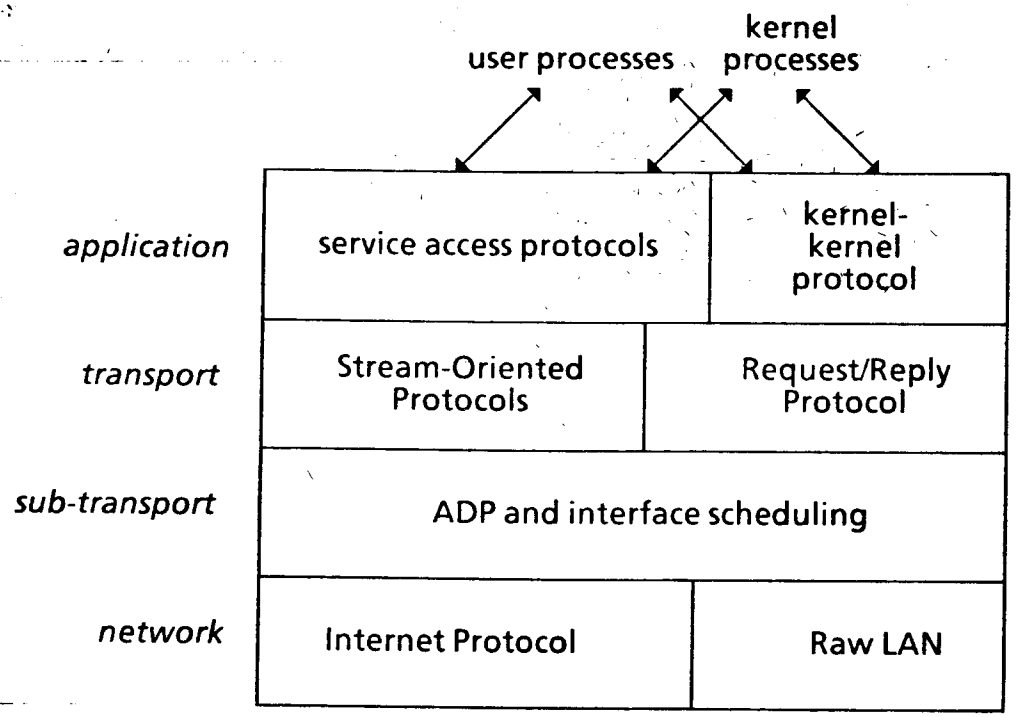


Figure 2: The DASH Communication Architecture.

8).

All RRP messages have a *kernel-level* part and may also have a *service-level* part. The format of the kernel-level part is determined by the message type and by the data representations of the two hosts. For machines of the same type, the native data representation is used. If the machines are of different types, a field in the header specifies whether the data is in the representation of the source, the destination, or in the standard representation. DASH uses the Sun External Data Representation (XDR) as the standard representation.

- The canonical execution environment and the canonical kernel are both based on a local message-passing facility, described in section 8. This is not part of the network communication model; an alternate execution environment might not use message-passing at all.

6.4.1. DASH Session-Oriented Communication

As discussed in section 6.2, a session-oriented interface to a remote service might involve several data channels, each with different reliability and performance needs. On the other hand, it must also be possible to treat a session as a unit for purposes of authentication and relocation.

This motivates the DASH organization for stream-oriented communication, which is a generalization of Ritchie's "stream" architecture for UNIX I/O [34].

- A *stream* is a half-duplex message channel.
- A *bundle* is a (possibly dynamic) set of streams between the same pair of hosts, and a set of protocols involving those streams. Either end of a bundle can potentially be moved to a different host.

Bundles may contain two types of protocols: *network protocols* perform functions (such as sequencing and retransmission) needed to compensate for network characteristics, and *functional protocols* perform tasks (such as terminal line disciplines) that are needed even for local bundles. Network protocols are *virtual* in the sense that they are realized only if the bundle endpoints are on different hosts. In addition, network protocols are symmetric around the network boundary (see figure 3).

Bundles and bundle endpoints are created as side-effects of RRP operations such as opening a service, passing a bundle endpoint, process creation (section 8.1), and splicing/tapping (section 8.2.3).

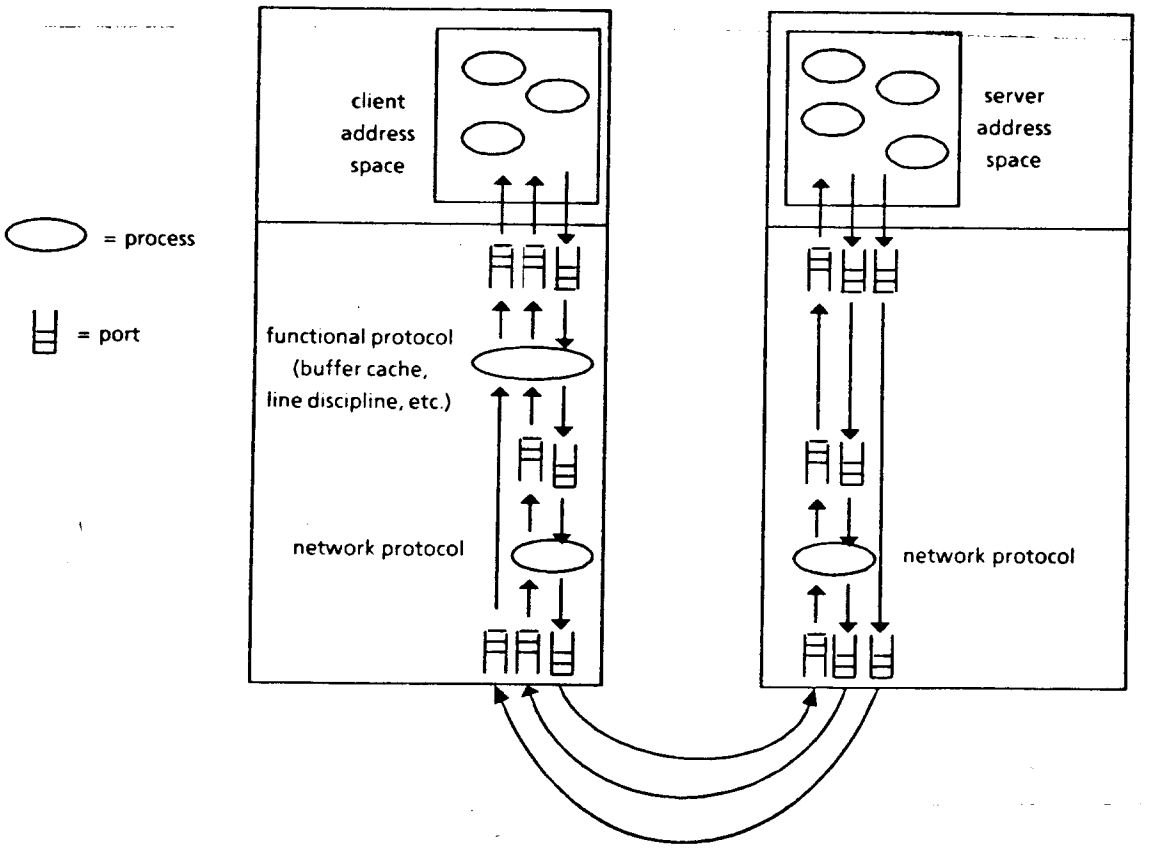


Figure 3: A DASH Bundle.

6.4.2. Network Security in DASH

DASH implements the decisions stated in section 6.3 by routing all network traffic through an *authenticated datagram protocol* (ADP); see [5] for a more complete discussion of ADP.

The ADP module on a host maintains a set of secure channels to other hosts. These secure channels may be implemented in different ways, depending on the security properties of the intervening network. In general, they use encryption or encrypted cryptographic checksums. If the network has the broadcast properties of Ethernet, a more efficient scheme that avoids cryptographic checksum can be used. If the network is assumed by both hosts to be physically secure and free of eavesdroppers, no encryption is used.

For each secure channel, ADP maintains lists of owners authenticated to and from the other end (see figure 4). This authentication uses PKE-based certificates. It is done only the first time a user communicates to a particular remote host, thus reducing encryption overhead.

7. SERVICES

A *service* is a logical resource in a client/server system. A *server* is a particular host or process that provides a service. Service access can be divided into two parts, each of which has both abstract and concrete components:

- The *service framework* is common to all services. Its abstraction may include naming, authentication, and reliability semantics. It is realized by the service-independent part of access protocols, which can be handled at the kernel level at both the client and server.
- A *service type* is the abstraction offered by a particular service, and is realized by service-specific access protocols.

What is the appropriate abstraction for the service framework of a VLDS?

The service framework abstraction should allow replication (multiple servers per service).

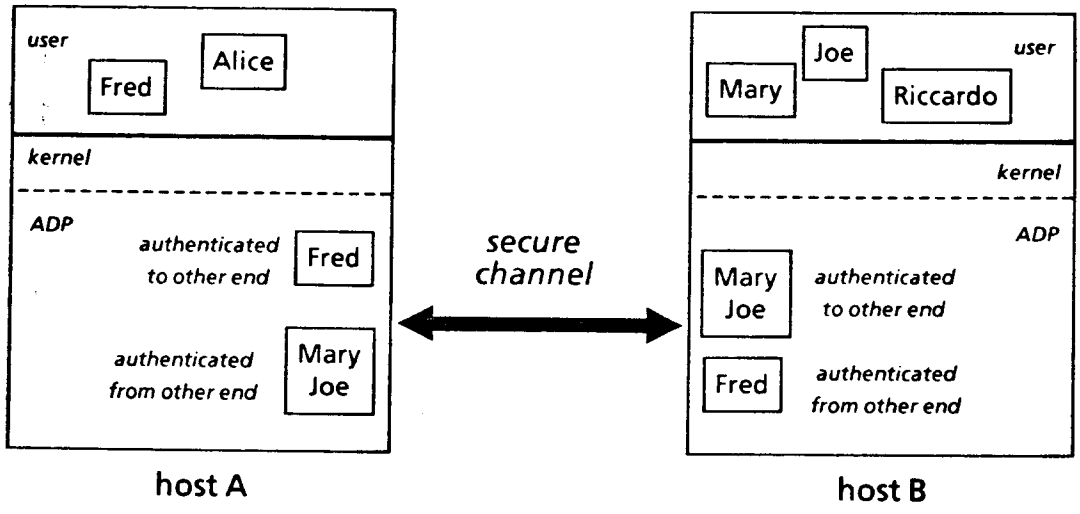


Figure 4: Authentication Caching in ADP.

Rationale: replication is an important means of increasing the availability, reliability and performance of services. It should be possible for this replication to be transparent to users. The alternatives then are to put replication at the client level, or to put it in the service framework. The latter alternative has many advantages; for example, recovery from a server crash can be made transparent to clients, assuming that the service framework is at the kernel level.

A service uses an *access protocol* that consists of a set of request/reply message types and perhaps a set of session-oriented protocols. Some systems [33] require that all messages be *typed* in a common specification language, and assume that messages are produced by automatically-generated *stub* routines.

The service framework should not include any notion of type.

Rationale: typing is language-dependent and can be done efficiently at the language level. Therefore by the principle of Minimal Assumption it should be done at that level.

Intra-service naming should be opaque to the service framework.

Rationale: services should be free to provide their own name-space structure, name resolution methods, and name-space consistency for replicated services.

Failure atomicity and concurrency control should not be in the service framework.

Rationale: they may not be needed. Services that do need transaction support often have properties that mandate a particular form of recovery processing or concurrency control. A VLDS should provide a framework for experimentation in this area, rather than attempting to provide a general solution. Projects such as TABS [42] and Clouds [3] have shown that a transaction mechanism can be built at the user level.

Protection schemes in a client/server system require that the client supply a capability or authenticated user name at some point. The server or its kernel allows or disallows accesses on this basis. Some protection schemes allow a user to have different *roles*, perhaps different groups to which the user belongs. There are two different ways of managing this [39]: either the client sends a list of roles with the request, or the client sends a "basic" ID that the server "expands" into a list of roles.

Protection should be done at the service level, based on system-level authentication of a single user ID.

Rationale: the principle of Minimal Assumption requires putting protection at the service level; authentication is the minimal system support required for this. Handling "multiple role" mechanisms in services rather than in the kernel is preferred for the same reasons.

7.1. The DASH Service Framework

The DASH service framework is designed to accommodate a large class of resources. It consists of a standard way of locating a service, means for communicating with services, provision for service migration and resilience, and provision for services to transfer their authorization to another service.

Services may be implemented by dedicated hosts, or by user- or kernel-level processes on general-purpose hosts. Kernel-level processes are used to provide access to physical devices.

7.2. Service Access

DASH provides two modes of service access: request/reply and session-oriented. In request/reply mode, the client supplies a pathname, a request message, a "need reply" flag, and a reliability level. The operation is invoked using the DASH RRP facility.

In session mode, the client kernel uses RRP to set up a session with an instance of the service. A bundle between the client and the service is created; the service specifies the number of streams in the bundle and the configuration of protocols.

In the DASH canonical execution environment, user-level services are typically provided by a group of processes sharing a single address space. Services can be accessed by user processes (via system calls) or kernel processes.

7.2.1. Replication

User-defined services can be replicated; the name-service entry for a service resolves to a set of (host name, service ID) pairs. This replication can be used in three ways:

- Location: in either request/reply or session mode, the kernel must locate an instance of the service. The kernel may attempt to use the closest or least-loaded instance. Alternatively, a service instance may forward the request to another instance because of its load or the distance to the client. In either case, the kernel can cache the network address of the instance, so that the next location is faster.
- Load-balancing of a session: at any point in a session-oriented service access, the service can tell the client's kernel that it wants to move its end of the bundle to another service instance, because it has become overloaded, because it is about to fail, or because it is accessing resources that are closer to the other instance.
- Crash recovery: when a service instance crashes, the client kernel can attempt to locate another instance an continue existing sessions.

There is no built-in support for data consistency or concurrency control among instances; this can be provided by special-purpose protocols, as in ISIS [6].

7.2.2. Service Ownership

Each service has a single owner, whose name is included in the name service entry for the service. All instances of the service are either user processes with that owner, or are run under the control of kernels with that owner.

7.2.3. Portals

The DASH service abstraction allows one service to provide naming and authorization on behalf of another. A service X can contain a *portal* to a service Y, in which case an access to X can be redirected to Y.

If the service Y does not do authorization directly, it must contain a hardwired list of services that contain portals to it. This list is maintained by Y's kernel. When a client accesses Y, it passes Y's kernel the name of the service that was used as a portal, and the whole pathname. Y's kernel then contacts that service and asks if the client is authorized for the service. The kernel also periodically rechecks this authorization; the period of reauthorization is determined by the service, either in its registration at the kernel or dynamically.

This authorization may also be cached in the service's kernel. As a result of this caching together with name translation caches in the client's kernel, it is possible for request/reply access to a service via a portal to use only two messages, in spite of the (logical) involvement of many services.

An example of the use of this mechanism is a *canonical file service* containing portals to other services. The file service already contains naming and authorization systems; the portal mechanism makes these available to other services, thereby simplifying those services.

7.2.4. Authorization

A service learns the name of its client on every access; this is part of the operation or session-open request message. The service can do *authorization* based on this name in whatever way it wants. The authorization scheme used by the canonical file service is based on access lists. A list can contain individual owners, owner subtrees (i.e. all owners whose names have a certain path prefix) or *groups*. Group membership is managed by *group services*, each of which may manage many groups. Group names are path extensions of the names of the group services.

The authorization schemes in the file service and the portal mechanism are designed to support the principle of *eventual revocation*: if a change in access list, group membership, or PKE key pair occurs, it may not have an immediate effect, but it will have an effect within a bounded amount of time. We believe that eventual revocation is an important property for VLDS services to have, since its absence implies a loss of autonomy.

7.2.5. Service Registration

A local system call registers a service on a host; it returns a local service ID (to be given to the name server). It can also give a list of portals to this service; each consists of a pathname and an authorization policy (reauthorization interval, and whether to reauthorize at each access). Reauthorization failures are reported to the server as exceptions.

7.2.6. Service Examples

The following services are used by the DASH canonical execution environment (the name services are used by all execution environments):

- *Name services*: each node in the global name space is managed by a service. Each kernel has at least one hard-wired entry in its service address cache that refers to a name service (usually the service for the parent of the host's pathname).
- *File service*: the canonical file service provides UNIX-like semantics (byte-array files, no locking or recovery, access-list authorization). It is used for booting, program loading, portals, and virtual memory backing store.
- *Group services*: described above.
- *Password service*: this is used as part of a login procedure in which the user types a password and the name of a password service. The login program then obtains the user's secret key from the password service by giving it the password. The secret key is then given to the local kernel, which uses it for subsequent authentication.

8. EXECUTION ENVIRONMENTS

The basic requirement of hosts in a distributed system is that they obey the network protocols; their internal organization is not important. It is possible that some hosts are single-address-space personal computers while others are multiprocessors with complex virtual memory hardware. Some hosts may implement only portions of the protocols; for example, a dedicated file server might ignore (or turn down) remote requests to create processes.

If the system is to support distributed programming or remote execution, however, hosts can no longer be viewed as black boxes. Instead, a host is seen as providing an *execution environment*, determined by its hardware and software. Loosely speaking, a host's execution environment is the mechanism and semantics for running programs on that host. Among the components of an execution environment are:

- The parameters of the remote process creation request. These might include a pathname of a program file or a handle on an existing service connection from which the program can be read.

- The interpretation of the program, i.e. the program file format, the processor's instruction set and the machine architecture.
- The abstractions provided to the program, such as the virtual memory, process, and local IPC models.
- The mechanism by which these abstractions are manipulated (i.e. the system call mechanism).
- The mechanisms by which processes can be externally controlled and monitored.

Two execution environments offer the same *execution abstraction* if they differ only in processor type and system call mechanism. These differences can be hidden by compilers and run-time systems and made invisible to the application programmer. To facilitate large-scale distributed computation, it is desirable that many hosts offer the same execution abstraction. However, it is impossible to formulate a single abstraction that exploits the potential of all hardware architectures.

A VLDS should define a canonical execution abstraction but should provide an open framework for others.

Rationale: this approach will encourage large-scale distributed computation, while making it possible for hosts with unusual architectures to be first-class members of the VLDS.

8.1. The Canonical Execution Abstraction

We now discuss desirable properties of the canonical execution abstraction (CEA) in a VLDS.

The CEA should provide separate kernel and (multiple) user address spaces.

Rationale: because of the need for load-sharing, it is desirable to provide the owner of a workstation with the guarantee that processes belonging to remote owners cannot destructively interfere with either the system or with other user processes. Given a general (unsafe) language model, this requires having separate address spaces for the kernel and for users.

Communication between different user address spaces, and between a user address space and the local kernel, should be based on message passing rather than shared memory or protected procedure calls.

Rationale: the pure message-based approach provides advantages for debugging: since a process can be "encapsulated" to an arbitrary extent by redirecting some or all of its messages [17]. It also facilitates process migration, since processes are not tied to stationary resources [46].

8.2. Process Control

Process control facilities (facilities for creating, stopping, starting and terminating processes, exception handling, and so on) are part of an execution environment. These facilities serve as the basis for:

- Shell functions such as command execution and job control.
- Distributed programming systems.
- Local debugging.
- Distributed debugging and monitoring.

High-level process control structures (such as process grouping, placement and migration mechanisms) should not be in the CEA.

Rationale: the principle of Minimal Assumption applies, since programming systems have varying process control requirements. For example, if a process grouping mechanism is to

handle multiple owners, it must define and enforce an authorization scheme. Putting such mechanisms at the user level gives increased flexibility [8] and is acceptably efficient because process control operations are relatively infrequent.

The CEA should optimize remote process creation.

Rationale: remote process creation is an inherent bottleneck in large-scale distributed computation. Its efficiency imposes a limit on the range of computation granularities for which parallelization can improve throughput. To allow a wide range of granularities, fast remote process creation must be possible. In particular, process creation should not necessarily involve a user-level server at the remote host; this involvement is required, for example, by Amoeba [32],

Network transparency at the system-call level should not be pursued.

Rationale: load-balancing, distributed computation, and data replication must be kept at the user level, and these demand network non-transparency. This does not imply that library and user-interface levels should not be network transparent.

8.3. Execution Environments in DASH

DASH defines a *canonical execution abstraction* while at the same time providing an open framework for other execution abstractions. Execution abstractions and kernel types are named (section 5.4.2). The name service entry for a host includes the names of its kernel type and of its architecture type. Remote process creation requests include an execution abstraction name and a machine architecture name; the request will succeed only if the target machine is capable of supplying the needed environment.

8.3.1. Virtual Memory in the DASH CEA

A *user address space* is shared by a set of processes, which all see the same virtual space. The processes in an address space are independently scheduled; on a multiprocessor more than one may run simultaneously. Page fault handling and backing store are normally provided by the kernel. However, there is a provision for user processes to receive exceptions on read or write references to specially-marked pages. This supports a wide range of possible user-defined schemes for local and global shared memory and single-level store.

All address spaces on a host contain a read-only *shared library segment.* This segment contains multiple libraries to which programs can be dynamically linked. There is no other provision for memory sharing between address spaces.

8.3.2. Message-Passing in the DASH CEA

The DASH local message-passing facility is related to those of the iAPX432 [12] and of Accent [33]. In DASH, a *port* is a FIFO mailbox that can be flow-controlled in terms of either number of messages or amount of data. All port structures are kept within the kernel address space; user processes refer to ports by temporary indices assigned by the kernel. Each port is *owned* by at most one user address space.

Local message-passing facilities are used by processes at both kernel (section 9.1) and user levels. For user processes, the operations are implemented as traps. Message operations are the only primitives available to user processes; all "system calls" are implemented as messages to the kernel.

Read and write operations can be done in any of the following modes:

- *Blocking*: the caller waits until the operation can be completed.
- *Non-blocking*: the operation returns immediately if the operation cannot be performed at the time of the call.

- *Reply*: a second *reply port* and an optional reply message are supplied. If the operation cannot be performed immediately, the call returns. If the operation later becomes possible and the reply port still exists and is writable (under its flow control) then the operation is performed and the reply message is written to the reply port. This mode can be used to allow a single process to multiplex several input and output ports, analogous to the BSD UNIX *select* system call [51].

There are two additional message operations: a *request/reply* operation involving a single port, and a *timed message* operation requesting that a message be written to a port after a real-time delay.

Null messages may be used in all operations. Together with flow control based on number of messages, this allows the message-passing system to provide synchronization abstractions equivalent to semaphores, monitors, and so on. Because the overhead of message allocation and formatting is eliminated, the message-based implementation is as efficient as a direct implementation.

User processes can use the local message-passing facility in the following ways:

- Processes in the same address space may use it to communicate or to synchronize access to shared memory structures.
- Processes in different address spaces can pass messages; two ports are used, and the message operations automatically transfer writes from one port to the other.
- Processes use ports to make system calls (section 8.3.3).
- Processes use ports for stream-oriented communication with services.

8.3.3. Process Control in the DASH CEA

Each user address space has two distinguished ports (see figure 5):

- The *kernel port* is used to deliver "system calls" to the kernel, normally in request/reply mode.

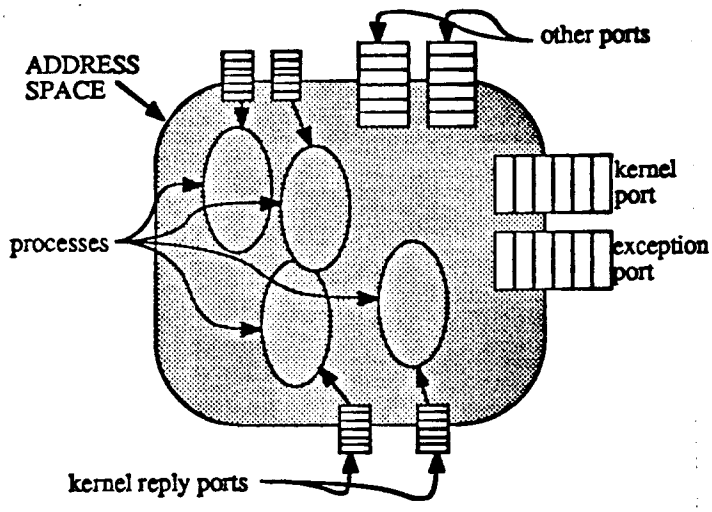


Figure 5: A DASH Address Space and Process Control Ports.

- The *exception port* is used to communicate exceptional conditions to process in the address space. These conditions include 1) illegal memory references, floating-point exceptions, and other synchronous exceptions, and 2) termination of a bundle because of a host crash, process crash, loss of authorization, or termination by the peer.

Debugging and monitoring of processes is supported by the following mechanisms (see figure 6):

- *Tapping* allows a process to non-obtrusively monitor all communication at a stream endpoint. When a message is written to a stream endpoint port, it is automatically written also to a *tapping stream* that terminates in a user-level *monitoring process* (possibly remote). The flow control requirements of the tapping stream are superimposed on those of the original port.
- *Splicing* allows the insertion of a user-level *filter process* (possibly remote) into an existing stream. If all the streams of a process are spliced, the process is *encapsulated*.

A monitoring or filter process can also set a *communication breakpoint* so that an user operation on a spliced or tapped stream causes the user process to block until it is released by the monitoring or filter process. A detailed design of these facilities can be found in

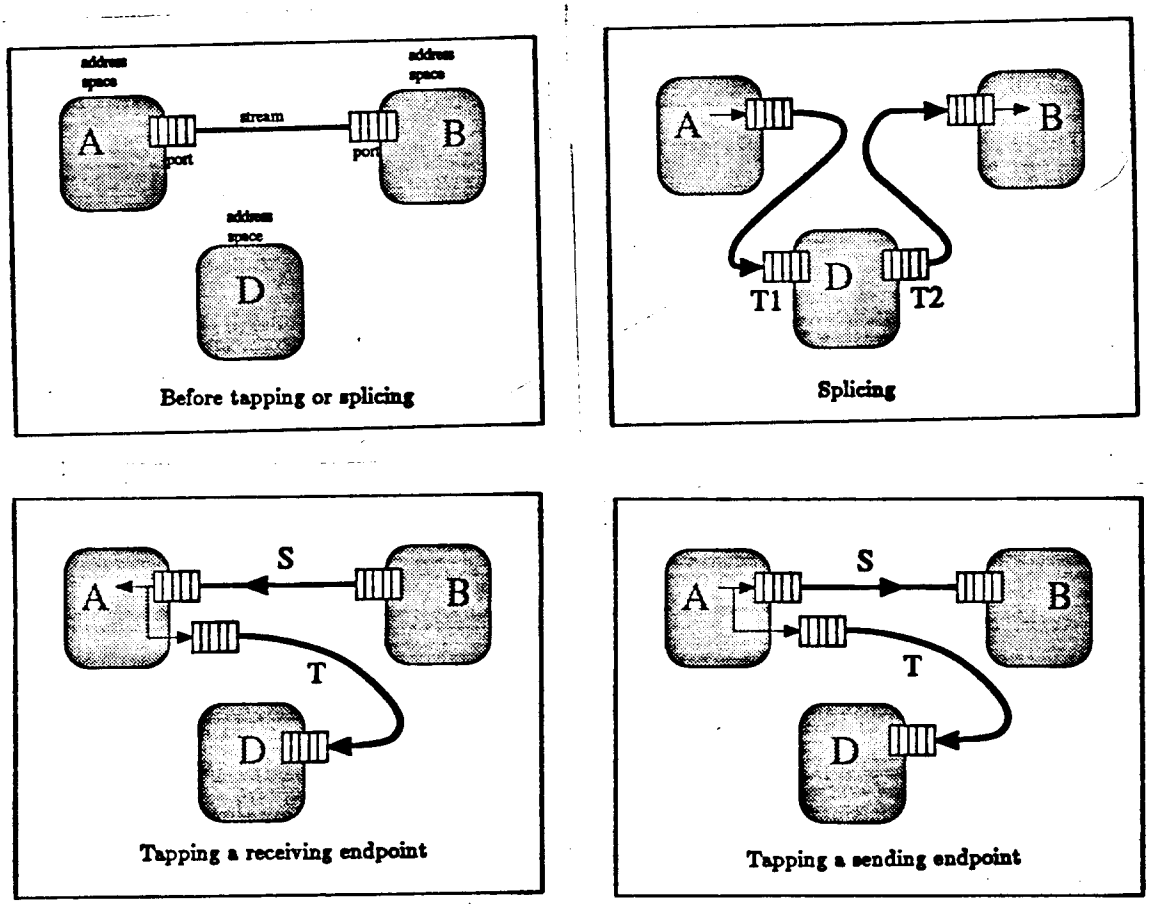


Figure 6: Tapping and Splicing.

[17].

8.3.4. Process Ownership in the DASH CEA

Each user address space has an owner, and in addition can *certify* itself for other owners by supplying their private keys. The remote process creation operation can specify the owner name of the process to be created; this caller must be certified for this name. This is designed to reduce the security problems associated with large-scale distributed computing. A user wanting to perform a distributed computation can create a new owner, put that owner on the access lists of only those resources needed for the computation, and create processes under the ownership of the new owner.

9. KERNEL STRUCTURE

The preceding section discussed execution environments in a VLDS, and described the DASH design. In practice, it is likely that the canonical execution environment will be implemented by a single kernel program (the *canonical kernel*) that will be ported to various machines by instantiating its machine-dependent parts. In this section we discuss desirable properties of the *canonical kernel structure* (CKS).

> **The CKS should promote kernel parallelism by 1) using parallel processes to perform kernel tasks when possible; 2) minimizing the serialization of these processes because of contention for resources.**

Rationale: much of the processing in a typical VLDS workstation is done in the kernel; examples include protocols, encryption, physical device drivers, and so on. Therefore, to efficiently utilize shared-memory multiprocessors, it is important that the kernel promote parallelism.

> **The CKS should minimize CPU bottlenecks for network communication bandwidth, such as data copying.**

Rationale: host CPU's are already the limiting factor in communication over LAN's. They will become "tighter" bottlenecks as networks become faster.

9.1. Structure of the Canonical DASH Kernel

The canonical DASH kernel consists of a dynamic set of processes that communicate both through the local message-passing facility (section 8.3.2) and through shared data structures. Kernel parallelism is promoted in the following ways:

- Each system call and remote operation is performed in a separate process; these processes can execute concurrently.
- Protocols are concurrent processes that can execute in parallel, thus pipelining a stream's protocol processing.
- ADP uses multiple processes to do encryption in parallel, potentially further pipelining network communication.
- Device drivers are processes and can execute in parallel with other processes.
- The concurrency control on kernel data structures is designed to reduce contention. Multiple lock types (spin locks, sleep locks, read/write sleep locks) are used. Locks are on data, not code.

A potential problem with the DASH design is that measures designed to improve performance on machines with large numbers of processors may hurt performance on uniprocessors. In particular, an optimal uniprocessor design would use procedure calls in many places where DASH uses message passing.

We contend, and hope to experimentally verify, that message-passing can be almost as fast as procedure calls, even on a uniprocessor. To accomplish this, the basic operations (memory allocation, locking, queueing, scheduling, context switching) must be fast, and kernel process communication must be treated as a special scheduling case.

10. SOFTWARE ENGINEERING

The canonical kernel of a VLDS system will have to be ported to diverse processors and system architectures. Hence the canonical kernel:

- Should be written in a language for which there exist compilers on a wide range of machines.
- Should have a machine-dependent part that is clearly delineated and is efficiently implementable on diverse hardware.
- Should be easy to maintain and modify.

The canonical kernel should be written in an object-oriented language

Rationale: it has been suggested [19] that the object-oriented programming style is best-suited to achieving the last 2 goals above.

The canonical kernel should not be written in a garbage-collected language.

Rationale: garbage-collected languages eliminate problems stemming from deleting memory objects to which there may be multiple references. However, there is an inevitable performance cost, and we believe that these problems can be solved in other ways in the DASH kernel. Therefore the principle of Performance dictates that garbage-collection not be used.

10.1. Software Engineering of the DASH Canonical Kernel

The DASH canonical kernel is being written in C++, an object-oriented extension of C [24], [43]. C++ was selected because it satisfies the above criteria and is supported in our current UNIX environment. The DASH kernel consists of a static set of *classes* (modules), and dynamic sets of *objects* and *processes*, all sharing a single address space. There are no global variables or data structures.

The DASH development environment makes heavy use of a remote symbolic kernel debugger, **kdbx**, based on the UNIX **dbx** debugger. **Dbx** interacts with the target process via the UNIX *ptrace* call, which can be used to access the process's memory space and control its execution. **Kdbx** runs on a UNIX host and communicates over a serial link with the DASH host being debugged. Calls to *ptrace* have been replaced with routines that send commands to the PROM monitor of the DASH host (currently a Sun 3/50).

11. CONCLUSION

We have described several system design areas in which the requirements of very large systems differ substantially from those of small systems. In each area we have identified and rationalized a set of design principles, and have suggested components of a design (that of the DASH system) that embody these principles.

The current status of the DASH implementation is as follows: kernel-level process scheduling and message-passing are complete, as is the ADP portion of the sub-transport layer. We are currently conducting experiments to measure the performance of ADP mechanisms given our current hardware (Sun 3/50 with AMZ8068 DES encryption chip, 10 Mb/s Ethernet).

Our future research plans include investigation of the following:

- Exploring the limits of size and granularity in distributed computing, and identification of possible bottlenecks (process creation, name resolution,

authentication, network latency).

- Utilization of multiprocessors: what performance gains will we get from kernel parallelism, and how does performance depend on processor scheduling, locking mechanisms, and locking granularity?
- Assessing the usefulness of the DASH bundle mechanism for long-distance high-performance services.
- Completion of the design for the sub-transport layer, and assessing its use for real-time communication applications such as digital audio.
- Exploring the limits of network performance with very fast ($>$ 1 Gb/s) networks and network interfaces.

12. ACKNOWLEDGEMENTS

Brian Bershad and Peter Danzig participated in the early phases of the DASH design; Luis-Felipe Cabrera and Roger Haskin provided useful feedback. Kevin Fall and Bruno Sartirana have been involved in the implementation of the prototype kernel.

13. REFERENCES

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian and Michael Young.
Mach: A New Kernel Foundation for UNIX Development.
USENIX Summer Conference Proceedings, pages 81-92, June 1986.

[2] Mark Hill et. al.
Design Decisions in SPUR.
IEEE Computer, pages 8-22, November 1986.

[3] J.E. Allchin and M.S. McKendry.
Synchronization and Recovery of Actions.
Operating Systems Review, 19(1):32-45, January 1985.

[4] Guy T. Almes, Andrew P. Black, Edward Lazowska and Jerre Noe.
The Eden System: A Technical Review.
IEEE Transactions on Software Engineering, 11(1):43-59, January 1985.

[5] David P. Anderson and P. Venkat Rangan.
A Basis for Secure Communication in Large Distributed Systems.
Submitted for publication, December 1986.

[6] K. Birman.
Replication and Fault-Tolerance in the ISIS System.
Proceedings of the 10th Symposium on Operating System Principles, Operating Systems Review, 19(5):63-78, December 1985.

[7] A. Birrell and B. Nelson.
Implementing Remote Procedure Calls.
ACM Transactions on Computer Systems, 2(1):39-59, February 1984.

[8] Luis-Felipe Cabrera, Stuart Sechrest and Ramon Caceres.
The Administration of Distributed Computations in a Networked Environment.
Technical Report No. University of California, Berkeley/Computer Science Department 86/268, Univ. of Calif., Berkeley, Spring 1986.

[9] David R. Cheriton.
The V Kernel: a Software Base for Distributed Systems.
IEEE Software, 1(2):19-43, April 1984.

[10] David R. Cheriton.
VMTP: A Transport Protocol for the Next Generation of Communication Systems.
1986 SIGCOMM Symposium, pages 406-415.

[11] David D. Clark, Mark Lambert and Lixia Zhang.
NETBLT: A Bulk Data Transfer Protocol.
DARPA Internet RFC-969, December 1985.

[12] George W. Cox, William M. Corwin, Konrad K. Lai and Fred J. Pollack.
Interprocess Communication and Processor Dispatching on the Intel 432.
ACM Trans. on Computer Systems, 1(1):45-66, Feb. 1983.

[13] D.H. Craft.
Resource Management in a Decentralized System.
Proceedings of the 9th Symposium on Operating System Principles, Operating Systems Review, 17(5):11-19, December 1983.

[14] T.P. Dobry, A.M. Despain and Y.N. Patt.
Performance Studies of a Prolog Machine Architecture.
12th International Symposium on Computer Architecture, 1985.

[15] Daniel D. Gajski and Jih-Kwon Peir.
Essential Issues in Multiprocessor Systems.
IEEE Computer, pages 9-28, June, 1985.

[16] G. Gawrys, P. Marino, G. Ryva and H. SHulman.
ISDN: Integrated Network/Premises Solutions for Customer Needs.
IEEE Int. Comm. Conf., pages 2-6, June 1986.

[17] Phillip Gibbons, Luigi Semenzato and Christopher Williams.
Debugging Support in a Distributed System.
Unpublished paper, December 1986.

[18] David K. Gifford, Robert W. Baldwin, Stephen T. Berlin and John M. Lucassen.
An Architecture for Large Scale Information Systems.
Proceedings of the 10th Symposium on Operating System Principles, Operating Systems Review, 19(5):161-170, December 1985.

[19] A. Goldberg and D. Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[20] Zimmermann, H.
OSI reference model- the ISO model of architecture for open systems interconnection.
IEEE Trans. on Commun., COM-28:425-432, April 1980.

[21] Roger Haskin.
Private communication, June 1986.

[22] W.D. Hillis.
The Connection Machine.
MIT Press, Cambridge, Mass., 1985.

[23] F. Hoornaert, J. Goubert and Y. Desmedt.
Efficient Hardware Implementation of the DES.
Proceedings of CRYPTO 84, pages 147-173, August 1984.

[24] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, 1978.

[25] Butler Lampson.
Hints for Computer System Design.
Proceedings of the 9th Symposium on Operating System Principles, Operating Systems Review, 17(5):33-48, October 1983.

[26] Keith Lantz, Judy L. Edighoffer and Bruce L. Hitson.
Towards a Universal Directory Service.
Operating Systems Review, 20(2):43-53.

[27] Ming-Kang Liu, David Messerschmitt and David Hodges.
An Approach to Fiber Optic Data/Voice/Video LAN.
IEEE INFOCOM 86, pages 516-523.

[28] Bob Lyon.
Sun Remote Procedure Call Specification.
Sun Microsystems, Inc. Technical Report, 1984.

[29] M.K. McKusick and M. Karels.
Performance Improvements and Functional Enhancements in 4.3BSD.
Summer USENIX Conference, pages 519-532, 1985.

[30] Paul Mockapetris.
Domain Names - Concepts and Facilities.
DARPA Internet RFC 882, November 1983.

[31] Sape Mullender and Andrew Tanenbaum.
Protection and Resource Control in Distributed Operating Systems.
Computer Networks, 8(5,6):421-432, 1984.

[32] Sape Mullender.
Private communication, November 1986.

[33] Richard Rashid and George Robertson.
Accent: A Communication-Oriented Network Operating System Kernel.
Proceedings of the 8th Symposium on Operating System Principles, pages 64-75, December 1981.

[34] D.M. Ritchie.
A Stream Input-Output System.
Bell Laboratories Technical Journal, 63(8):1897-1910, October 1984.

[35] R.D. Sansom, D.P. Julin and R.F. Rashid.
Extending a Capability Based System into a Network Environment.
1986 SIGCOMM Symposium, pages 265-274.

[36] M. Satyanarayanan, John M. Howard, David A. Nichols, Robert N. Sidebotham, Alfred Z. Spector and Michael J. West.
The ITC Distributed File System: Principles and Design.
Proceedings of the 10th Symposium on Operating System Principles, Operating Systems Review, 19(5):35-50, December 1985.

[37] Michael D. Schroeder, Andrew D. Birrell and Roger M. Needham.
Experience with Grapevine: the Growth of a Distributed System.
ACM Transactions on Computer Systems, 2(1):3-23, February 1984.

[38] Stuart Sechrest.
Attribute-Based Naming of Files.
Unpublished paper, February 1986.

[39] Stuart Sechrest.
Issues Regarding User Roles in a Distributed System.
Unpublished paper, December 1986.

[40] S. Shimada, K. Nakagawa and I. Takeshi.
Gigabit/s Optical Fiber Transmission Systems - Today and Tomorrow.
IEEE Int. Conf. on Comm., pages 1538-1542, June 1986.

[41] Alfred Spector.
Implementing Remote Operations Efficiently on a Local Network.
Communications of the ACM, 25(4):246-260, 1982.

[42] Alfred Spector, Dean Daniels, Daniel Duchamp, Jeffrey Eppinger and Randy Pausch.
Distributed Transactions for Reliable Systems.
Proceedings of the 10th Symposium on Operating System Principles, Operating Systems Review, 19(5):127-146, December 1985.

[43] Bjarne Stroustrup.
The C++ Programming Language.
Addison-Wesley, 1986.

[44] Douglas B. Terry, Mark Painter, David W. Riggle and Songnian Zhou.
The Berkeley Internet Domain Server.

USENIX Summer Conference Proceedings, pages 23-31, June 1984.

[45] Douglas B. Terry.
Distributed System Support for Voice in Cedar.
ACM SIGOPS Workshop, Amsterdam, October 1986.

[46] Marvin Theimer, Keith Lantz and David Cheriton.
Preemptable Remote Execution Facilities for the V-System.
Proceedings of the 10th Symposium on Operating System Principles, Operating Systems Review, 19(5):2-12, December 1985.

[47] Tom Truscott, Bob Warren and Kent Moat.
A State-Wide UNIX Distributed Computing System.
Proceedings of the 1986 Summer USENIX, pages 499-513.

[48] Jonathan Turner.
Design of a Broadcast Packet Network.
IEEE INFOCOM 86, pages 667-675.

[49] Bruce Walker, Gerald Popek, Robert English, Charles Kline and Greg Thiel.
The LOCUS Distributed Operating System.
Proceedings of the 9th Symposium on Operating System Principles, Operating Systems Review, 17(5):49-70, November 1983.

[50] Brent Welch.
The Sprite Remote Procedure Call System.
University of California, Berkeley Tech Report, July 1986.

[51]
The 4.3BSD UNIX Programmer's Manual, Volume 1, 1986.