THE DASH PROJECT: AN OVERVIEW

David P. Anderson
Domenico Ferrari

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

February 29, 1988

ABSTRACT

The DASH project at UC Berkeley is studying problems arising in the design of large, high-performance distributed systems, and is building an experimental system. The system's major design goals are centered in three areas: 1) IPC performance, 2) global architecture, and 3) local architecture. In each of these areas, *vertically integrated* mechanisms are used to achieve design goals, while an *open system* structure is maintained where possible. This report describes the motivation and principles of the DASH project, and sketches the current design of the DASH system.

Sponsored by the California MICRO program, AT&T Bell Laboratories, Digital Equipment Corporation, IBM Corporation, Olivetti S.p.A, and the Defense Advanced Research Projects Agency (DoD) Arpa Order No. 4871. Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

1. INTRODUCTION

The name *DASH*[1] refers to:

(1) A research project studying the design principles of future distributed systems.

(2) A new distributed system architecture embodying the results of this research.

(3) An operating system kernel implementing the distributed system architecture.

This report is an overview of all three of these aspects of DASH. We describe first the motivation and goals of the project, then the distributed architecture and the kernel. An earlier report, *Issues in the Design of Very Large Distributed Systems* [4], expands on the motivation and principles of DASH. The DASH system design is described in more detail in three companion reports: *The DASH Communication Architecture* [33], *The DASH Virtual Memory System* [35], and *The DASH Local System Architecture* [34].

1.1. Motivation, Assumptions and Goals

Much current research in operating systems is focused on *high-level mechanisms* such as distributed transactions, support for replicated data, object-oriented programming systems, user interfaces, and facilities for parallel distributed computation. *Low-level mechanisms* such as virtual memory, process control, kernel structure, naming, local IPC, and network communication have not kept pace with the progress in high-level mechanisms. Many of the current research projects are based on outdated operating systems (such as UNIX[2]) and are crippled by the inappropriate low-level mechanisms provided by these systems.

The main research objective of DASH is the development of optimal low-level mechanisms for the next generation of distributed computer systems. We have taken the following steps towards this goal:

1) Extrapolate trends in computer and communication technology into the middle-to-distant future (5-20 years).

The dominant host type will be the workstation; many will be shared-memory multiprocessors with a small number (10-100) of processors [13, 15]. Communication networks based on fiber optics will allow low-delay (30 to 50 milliseconds coast-to-coast) and high-bandwidth (.01 to 1 Gigabit/second) communication between most pairs of hosts in the U.S., and eventually in the world [17, 26, 31].

2) Propose functions and facilities of future computer systems based on this technology.

We use the term *very large distributed system* (VLDS) [4] to refer to a hypothetical system running on the hardware base described earlier. A VLDS will link thousands or millions of hosts under diverse ownership. Its main function will be to provide secure, well-integrated access to logical *services*. These services might provide access to public databases (such as encyclopedias and archives), news media, sales, advertising, banking, interpersonal communication (mail, telephone, facsimile, and video conferencing), and entertainment (including distribution of audio

[1] DASH was originally an acronym for Distribution, Autonomy, Security and Heterogeneity, attributes we viewed as desirable in an operating system. This list kept growing, and rather than lengthen the name we kept it and removed its acronym status.

[2] UNIX is a trademark of Bell Laboratories.

and video).

The processing power of VLDS hosts (and perhaps of specialized compute servers) is another type of remotely-accessible resource. A VLDS must support load balancing and large-scale parallel computation; in the latter case, thousands or millions of processors might be involved in a single computation.

3) **Identify the basic system-level requirements of these functions and facilities.**

These requirements fall into three main groups: 1) IPC performance, 2) global system architecture, and 3) local system architecture. The groups are discussed in Sections 3, 4 and 5 respectively.

4) **Propose designs and mechanisms for satisfying these requirements.**

The DASH project is developing a design for a VLDS. Our current design is sketched in the remainder of this report, and is described in more detail in the companion reports ([33-35]).

5) **Study these mechanisms by implementing them and evaluating the resulting system.**

The DASH project is currently building an operating system kernel (the DASH kernel) that implements our distributed system design and will be used to evaluate and refine it. The kernel is being implemented on Sun 3 workstations, and will soon be ported to a Sequent Symmetry shared-memory multiprocessor.

In summary, the DASH project is building a foundation for very large distributed systems. Because of the synergy that may arise from combining communication, service access, and processing in a single unified system, VLDS design is an important direction in computer systems research. A VLDS will subsume the proposed functions of Integrated Services Digital Networks (ISDN) [14]. The use of a VLDS for high-performance computing will augment (and often replace) the use of specialized hardware [12], general-purpose parallel hardware [16], and supercomputers to address the processing requirements of graphics, artificial intelligence, simulation and scientific applications.

2. DASH DESIGN PRINCIPLES

The design of computer systems, and especially distributed systems, is often described in terms of *modules* i.e., logical components that interact only through abstract interfaces. Some of the modules are part of the protocol hierarchy; others are added by the local system architecture.

Likewise, the desired properties of the system are described by a set of *design goals*. In DASH, these properties fall mainly into the three areas mentioned earlier: IPC performance, global system architecture, and local system architecture. Each of these areas involves several of the modules, so we may represent the system design as a two-dimensional system of interactions, with modules on one axis and design goals on another (see Figure 1). We identify the following design principles arising from these multi-layer interactions:

Vertical Integration of Mechanisms:

Mechanisms must often span multiple system modules to achieve VLDS design goals. For example, high-performance IPC may be possible only by integrating mechanisms at the levels of virtual memory, process scheduling, and network

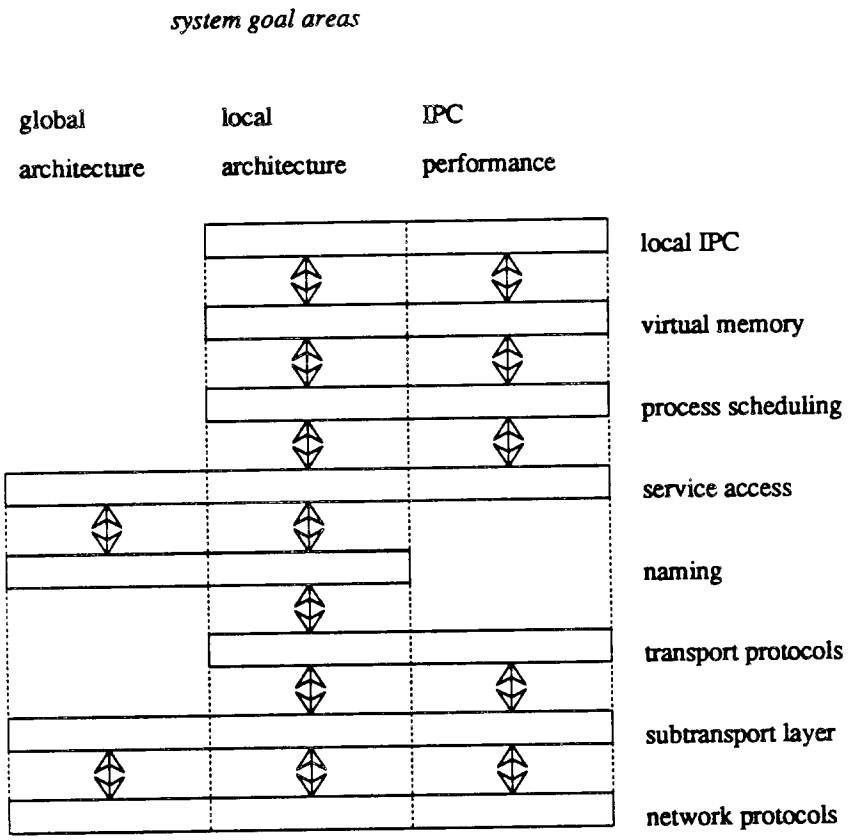

Figure 1: Interactions Between Design Levels.

communication.

Open Architecture:

A VLDS will encompass hosts with different hardware architectures, different application areas, and different computing paradigms. Therefore, the portions of the system that are *standardized* (i.e., those portions that a host must implement in order to take part in resource sharing) should be minimal. Basically, the standardized part of the system must provide naming, a way to move bytes securely and efficiently, and not much beyond that.

These two principles are not followed in some approaches to building distributed systems. For pragmatic reasons, most current systems are built on top of existing general-purpose protocol hierarchies (the V system [11] is a notable exception). Such hierarchies usually export a simple interface that precludes vertical integration. In addition, both principles dictate against building a VLDS as an extension of an existing centralized system (the approach of Mach [22]), or as an "interconnection" layer on top of existing

operating systems (as is done in Cronus [23]).

3. HIGH-PERFORMANCE REMOTE INTERPROCESS COMMUNICATION

A VLDS will provide a mechanism for *clients* to communicate with *services*. Some services (such as those based on digital audio and video) will require high-performance remote interprocess communication (IPC). The general-purpose communication mechanism of a VLDS must support such communication. This mechanism consists of several components:

(1) The movement of data across networks.

(2) Protocol flow control and reliability mechanisms.

(3) The movement of data between device interfaces (including network interfaces) and main memory.

(4) Scheduling, synchronization and processing time of the communicating processes, and of intermediate protocols.

(5) The movement of data between virtual address spaces on a single host. Services may run in separate address spaces. Access to a local service requires data movement between two user spaces, and access to a remote service requires data movement between user and kernel spaces at both ends.

To maximize performance, the remote IPC mechanism of a VLDS must address each of the above components, and their interactions. In particular, the following goals can be identified:

- To reduce the overhead of host processing, currently the bottleneck in most IPC systems. In particular, to minimize software copying, and to avoid software encryption and checksumming where possible.
- To provide real-time performance guarantees. Scheduling of communication resources (transmission queues in hosts and switches, and processing in clients and protocols) must be done on the basis of real-time deadlines.
- To support configurable stream protocols. Request/reply communication is inadequate for many high-performance applications over long distances. In addition, the reliability, flow control, network capacity, and integrity functions of stream-oriented IPC must be separated, allowing clients to use only the functions they need.
- To provide a high-performance security mechanism that 1) allows the user to specify the level of security desired, and 2) allows the presence of secure hosts and secure local networks to be exploited.
- To accommodate a variety of network architectures.

3.1. The DASH IPC Design

3.1.1. Real-Time Message Streams

In existing distributed systems, the network-dependent communication interface typically provides a simple abstraction such as unreliable, insecure datagrams. Higher software layers use this facility to provide higher-level abstractions such as reliable request/reply message-passing [10], reliable secure typed message streams [22], or reliable byte

streams [20]. This approach simplifies the task of porting the system to different network types. However, the simple nature of the basic abstraction (such as datagrams) does not allow communication clients to express their performance, reliability and security needs, or their workload parameters, to the communication provider. This makes it impossible for the provider to use the most efficient mechanisms or to provide real-time performance guarantees. It also makes congestion control in large networks difficult.

In an attempt to solve these problems, the DASH network communication system is based on an abstraction called *real-time message streams* (RMS) [5]. An RMS is a simplex communication channel between a *sender* and a *receiver*. Message boundaries are preserved and messages are delivered in sequence. In addition, an RMS has various parameters reflecting its performance and security properties. Specifically, it has the following Boolean parameters:

Authentication: if true, then impersonation (delivery of a message with incorrect source label) is impossible.

Privacy: if true, then eavesdropping (access to a message by a host or process other than that specified by the target label) is impossible.

An RMS has the following performance parameters:

Capacity: an upper bound (enforced by the sender) on the amount of data outstanding within the RMS at any point (i.e., sent but not yet delivered).

Maximum message size: an upper bound (enforced by the sender) on the size of individual messages.

Delay bound: message delay is the elapsed real time between the start of the send operation and the moment of delivery. An RMS has an upper bound (guaranteed by the RMS provider) on message delay. The components of the delay may include network transmission delay, queueing and processing delays at the sender and at intermediate switches, and processing at the receiver. This bound may be *deterministic*, *statistical*, or *best-effort*.

Average bit error rate: this parameter reflects the combination of 1) the error rate of the underlying transmission medium, and 2) the effectiveness of the checksumming algorithm. It is guaranteed by the RMS provider.

Average loss rate: this reflects the expected rate of packet discarding from buffer overrun and checksum failures.

An RMS creation request includes *desired* and *acceptable* parameter sets. The actual parameters of the resulting RMS are returned to the client. These parameters must be compatible with the request's *acceptable* parameters; the request is rejected if this is not possible. The RMS provider tries to match the *desired* parameters as closely as possible.

3.1.2. The DASH Network Communication Structure

In DASH, the RMS abstraction appears in the interface to the network-dependent part, and at higher levels of the system as well. RMS is the basis for a request/reply communication facility in which the RMS features serve to optimize request/reply performance. The structure of the DASH network communication system is shown in Figure 2.

A DASH system can encompass many networks. Each network has a set of protocols for implementing RMS between any two of its nodes. Note that, in this discussion, the term

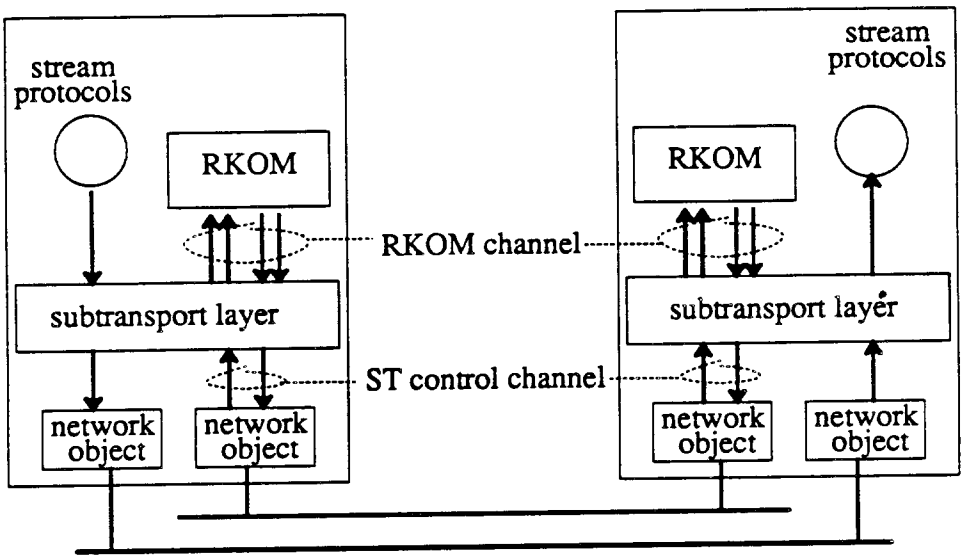

Figure 2: The DASH Network Communication Structure.

network refers to an abstract entity, not necessarily to a physical network. For example, the DARPA Internet (with the addition of RMS support) and a local Ethernet could be separate DASH networks, although they might share the same host interfaces and network media. The DASH *network layer* encapsulates everything below the network RMS abstraction: network-specific protocols for establishing RMS's, routing protocols, address resolution protocols, and so on.

The subtransport (ST) layer provides authentication and privacy, caching and multiplexing of network-level RMS's, piggybacking of messages, and other services [33]. The ST protocol must be implemented by all DASH hosts.

The ST 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 [7]. In general, they use data encryption or cryptographic checksums. On Ethernet-like networks, a more efficient scheme that avoids cryptographic checksumming of data 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, ST maintains lists of owners authenticated to and from the remote host (see Figure 3). This authentication uses public key encryption (PKE)-based certificates. It is done only the first time a user communicates with a particular remote host, thus reducing encryption overhead.

The *transport layer* consists of a set of protocols that use the ST facilities. One of these, the *Remote Kernel Operation Mechanism* (RKOM), is used for all request/reply communication, and is mandatory for all DASH hosts. The other transport protocols are

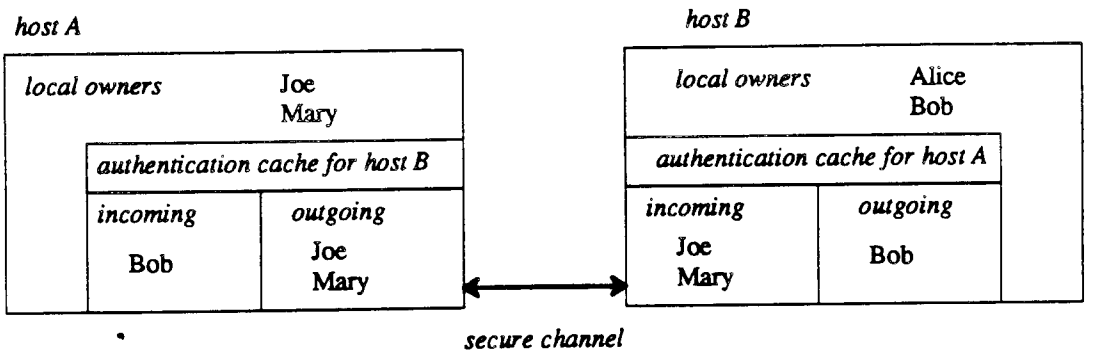

Figure 3: Authentication Caching in the Subtransport Layer.

stream-oriented. They are implemented as separate processes that can be dynamically configured, in the style of Ritchie Streams [21]. These protocols provide functions such as reliability, RMS capacity enforcement, and flow control.

DASH allows the RMS abstraction to span processes. A *subuser RMS* spans protocol processes. Its delay bound includes protocol processing time. A *user-level RMS* spans user processes. Its delay bound includes end-process CPU time as well. In both cases, the enforcement of the delay bound uses deadline-based scheduling of protocol or user processes.

3.1.3. RMS Examples

To see the importance of RMS parameters, suppose that a client (say a transport protocol serving a user program) requires data privacy. The protocol requests an RMS from the subtransport layer. The desired and acceptable parameter sets both have the *privacy* flag set. Depending on the network, the following cases are possible:

(1) Privacy is provided by data encryption in the subtransport layer.

(2) The network has link-level encryption hardware; the subtransport layer learns this (it is a property of network-level RMS's) and does no data encryption.

(3) The network is considered secure, so no data encryption is done.

In any case, the RMS parameters allow the subtransport layer to use the optimal mechanism for privacy. If a client does not require privacy, no mechanism is used (which is again optimal). Without the RMS security parameters, this optimization would not be possible. The situation is similar for data integrity. Based on the values of RMS parameters, the optimal checksumming mechanism can be determined.

The following examples illustrate the uses of the RMS capacity and performance parameters:

- Initial request and reply messages in a request/reply protocol use an RMS with low delay bound. The RMS capacity may be large, unless it is known that request or reply messages will be small and infrequent.
- A stream protocol for bulk data transfer uses a high capacity, high delay RMS for data. Reliability acknowledgements uses low capacity, high delay RMS's. Flow control acknowledgements uses a low delay, low capacity RMS.
- Digitized voice uses a high capacity, low delay RMS, perhaps with a statistical delay bound. A high bit error rate may be acceptable.
- Communication involving human user interface traffic (such as for network window systems [24]) can tolerate a moderate amount of delay because of human perceptual limitations. The RMS from user to application carries mouse and keyboard events, and can have low capacity; the RMS in the opposite direction carries graphic information, and requires higher capacity.

In all these cases the explicit specification of client needs increases the likelihood that the provider can accommodate them. For example, if packet queueing in an internetwork gateway is done using RMS-specified deadlines, then a low-delay packet can be sent before high-delay packets that would otherwise cause it to be delivered late. A network may be capable of providing low delay or high capacity, but not both. The RMS parameters allow the client to choose.

The use of RMS in DASH is based on anticipated needs and on projections of future network technology; the RMS abstraction is not supported on current networks, and cannot be built on top of simpler abstractions such as datagrams or virtual ciruits. However, we feel that our approach is necessary for exploiting the advances in communication technology that will occur in the near- and long-term future.

3.2. Movement of Data Between Local Address Spaces

The efficiency of moving a large amount of data between virtual address spaces (both user spaces and kernel space) on a single machine is a major component of IPC performance. Software memory copying is the straightforward way to move data between spaces. However, memory bandwidth is improving at a slower rate than processor and network speeds. Thus, memory copying is likely to be an IPC bandwidth bottleneck and a major source of IPC delay. In addition, the bus traffic generated by memory copying will degrade system performance on shared-memory multiprocessors. Virtual memory (VM) remapping, as opposed to memory copying, is an attractive approach to moving data. However, remapping in shared-memory multiprocessors can be costly because of the problem of translation lookaside buffer (TLB) inconsistency.

The DASH mechanism for local data movement [32] eliminates many of the overheads that would otherwise arise from VM remapping in shared-memory multiprocessors. Put simply, we reduce the need for synchronous unmapping, and, when such remapping is necessary, we do it efficiently.

The DASH VM system has an *IPC region* that is shared (although with different levels of protection) among all address spaces. All data to be moved between spaces are placed in the IPC region. The local user IPC system involves the following layers (see Figure 4):

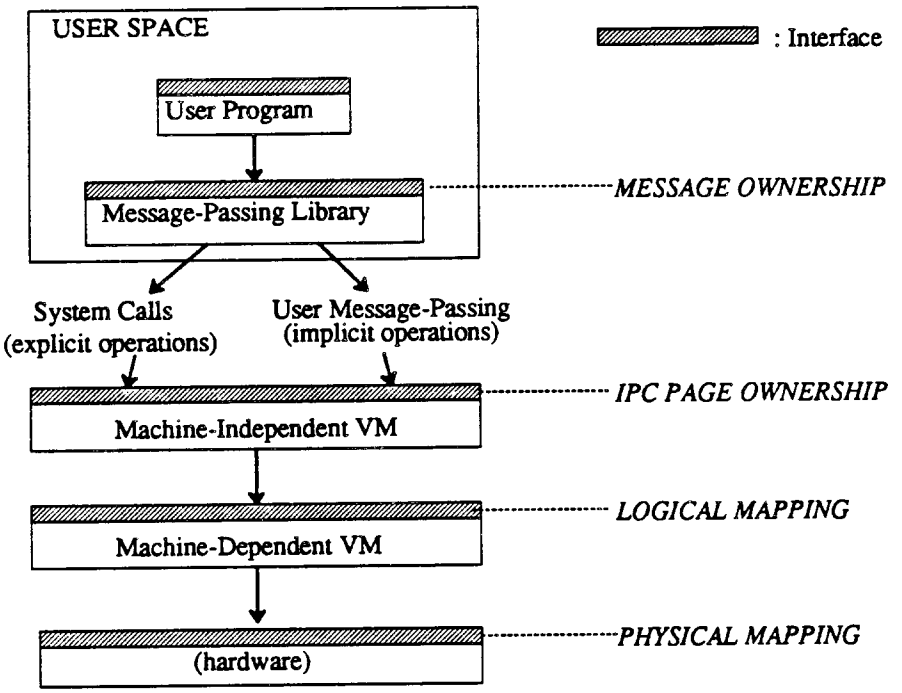

Figure 4: Logical Levels of Page Ownership and Mapping.

- A *message-passing (MP) system* providing operations to allocate, access, send, receive and deallocate messages. It is implemented by a user-level library that handles some operations itself and traps to the kernel for others.
- The facility for *protected shared memory* provided by the VM systems's IPC region. This facility defines a notion of *ownership* of pages in the IPC region, and is used by the MP system for data movement. The facility can also be used directly by user processes via system calls (which are themselves implemented as MP operations).
- The *Logical VM mapping* interface of the VM system, whose operations include 1) mapping IPC pages on a single CPU, and 2) unmapping IPC pages on a set of CPU's. The unmapping operation (which on some architectures may require interprocessor interrupts) may be either *synchronous* or *asynchronous*. The latter type is slower but has less total overhead since multiple operations can be batched.
- The *Physical VM mapping* function of the VM system; this is the machine-dependent implementation of the logical VM mapping operations.

In the DASH message-passing system, the semantics of send() are that the sender's ownership of the IPC pages in the message is transferred to the receiver. The sender may

have multiple ownerships of a page; when the last ownership is relinquished, the IPC page is unmapped from the sender's address space. The send() and receive() operations have parameters that can be used by the implementation to reduce work:

- The sender may specify whether it believes that the receiver trusts it. If so (and if the sender is correct), the unmap operation on send() can be deferred, and may never be done.
- The receiver may specify whether it trusts the sender or not (e.g., the kernel owner is always trusted). If the sender is not trusted, the receiver must wait until the operation of unmapping the page from the sender is completed (otherwise the sender could modify the data after it has been received). If the sender is trusted, it is not necessary to wait for a previous asynchronous unmap operation to complete.
- The receiver may specify whether to physically map in message pages immediately, or to map them in on demand. In the second case, a page is mapped in by the page fault handler when the receiver first accesses the page. No mapping is done if the page is not accessed. This optimization may be significant for applications that forward messages (e.g., a file service that receives a block from a disk device and sends it to the network without accessing it). Message forwarding is a common communication paradigm when services offered by the system execute at the user level.

The above optimizations can eliminate operations. When operations are necessary, our design allows them to be done efficiently. Synchronous unmapping is more expensive than asynchronous unmapping, and we avoid it when possible. For both types of unmapping operations, the VM system maintains a list of processors on which a page has been mapped, and only unmaps it from those processors.

3.3. DASH IPC: Summary

The DASH IPC design is *vertical integrated* in the following ways:

(1) The RMS abstraction spans all levels (network and local) of the IPC system, making real-time communication between user processes possible.

(2) The elimination of software copying in network communication involves a combination of message-passing semantics, protocols, kernel architecture, and virtual memory.

(3) Although authentication and security mechanism are defined at a high level (see the discussion of global naming in Section 4), they are enforced at a low level of the protocol hierarchy. This can substantially reduce the expense of these functions [6].

The DASH IPC facility is also an *open system*:

(1) The RMS interface allows different network types to make their nonstandard functions (e.g., encryption in hardware) visible to upper layers, and to implement RMS in a network-specific way.

(2) The framework for stream protocols allows new protocols to be added to the kernel, and combined with other protocols, in a standard way. Clients can use protocols that provide the exact combination of functions (capacity enforcement, flow control, and reliability) that they need.

The DASH IPC design has numerous advantages over those of current distributed systems. In systems such as V [11] designed for local-area networks, IPC is often limited to reliable request/reply communication. This is inadequate for VLDS. In systems such as Mach [1], IPC is forced to pass through a user-level "network server", and this further limits potential performance. IPC performance can be hampered by the basic semantics. For example, the semantics of the UNIX write operation (shared by Mach and most other systems) are that the sender retains a (logical) copy of the data sent. In DASH, the semantics are that the sender loses the data from its space. This eliminates the work of creating a logical or physical copy of the data.

4. GLOBAL SYSTEM ARCHITECTURE

The *global architecture* of a distributed system is centered in its naming mechanisms. There often are naming mechanisms at multiple levels; they differ in the nature of the named entities, and the means of assigning and resolving names. Some of these mechanisms may also involve authentication and security.

The global architecture for a VLDS has the following requirements:

- There must be a global naming system on which security functions are based.
- The naming system must support organizational *autonomy* in the senses of 1) hierarchical delegation of authority for name assignment, and 2) lack of central trusted agents in name resolution.
- Naming must be source- and target-location independent. This reduces the location-dependence of program execution, thus simplifying large-scale distributed programming [2].
- The naming system must be *scalable* so that performance does not decline with increasing system size, even when remote references are frequent [29].

4.1. The DASH Global System Architecture

4.1.1. Naming and Authentication

DASH global names are symbolic pathnames in a single tree-structured name space. There are four types of named entities in the DASH global name space: hosts, owners, services, and name services. The internal nodes of the tree represent name services, and the leaves of the tree represent the other entity types (see Figure 5). The different entity types, and their associated attributes, are as follows:

- An *owner* is an individual human user or "role". Its attributes include two public keys: a *user* key and a *kernel* key.
- A *host* is a network-level communication endpoint. Its attributes include a list of its network addresses and the name of its owner.
- A *service* is a logical resource provided by set of programs or processes. Its attributes include 1) a list of (host name, instance ID) pairs, each specifying an instance of the service, and 2) the name of the owner of the service.
- A *name service* is a special type of service that manages the names of other entities. A name service maintains a single *directory* in which each entry has a name (a pathname component), a type, and a set of attributes.

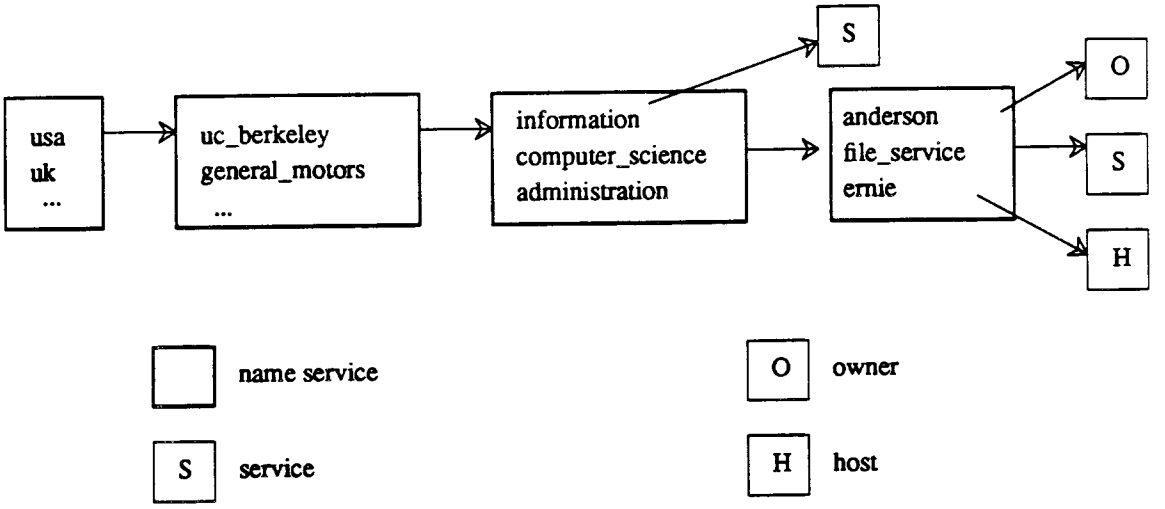

Figure 5: The DASH Global Name Space.

The DASH service access mechanism allows services to extend the global name space below their own name. Hence they can provide global names for the objects they manage. For example, a file service might provide hierarchical naming of its files, so that a reference to */usa/ucb/cs/fs/anderson/foo* might map to the file */anderson/foo* within the file service */usa/ucb/cs/fs*. This removes the need to distinguish the two levels of naming, and makes it possible for a service to serve as a manager of named "objects" or to provide logical "sub-services".

The DASH kernel maintains a cache of name resolutions. Even with this caching, the work involved in component-by-component resolution of long pathnames may be excessive, particularly if it must be done for frequently-accessed objects such as disk files. To avoid this problem, the DASH kernels allows user programs to obtain *name tokens*, each of which represents a pathname and its cached resolution. In further name references the client provides the token and a symbolic name extension; the kernel can begin resolution starting from the object represented by the token. (An analogous mechanism for references to names within services is described in the next section.)

4.1.2. Service Access

In the DASH distributed system architecture, *services* are a class of logical resources. A DASH *service* is a set of *instances* that together provide a logical resource. Each instance resides on a single host, and may consist of a process, a set of processes, or a "registration" with the host kernel that causes a process to be created as needed.

The intent of the DASH global architecture is that services, where possible, should be globally accessible. The service access facility described in this section makes remote

access, for both the client and server, as convenient as local access.

The DASH *service access mechanism* (SAM) allows clients to name and communicate with services in a uniform way. It provides:

Replication transparency: a client need not know which instance handles a particular request or session.

Location transparency: service names do not specify or limit the location of the servers.

Failure transparency: if a service instance fails, SAM may locate another instance of the service and connect the client to it automatically.

Protocol flexibility: services may provide interfaces that have real-time communication performance requirements, or that need special-purpose stream protocols.

A replicated service may provide a consistent data abstraction, in which case it needs to ensure data consistency between instances. DASH does not supply or dictate any method for this, or for ensuring the atomicity or permanence of operations on services. Such mechanisms must be supplied by the services themselves, perhaps in cooperation with a higher-level transaction manager.

Services can be accessed in two basic modes. In *Request/reply mode*, the operation is conveyed to the server via RKO; two reliability types (*maybe* and *exactly once*) are available. In *Session mode*, SAM locates an instance of the service and sets up a communication channel (or *bundle*) between the client and server.

A client may request a *service token* representing an object within the service. The token can thereafter be supplied in lieu of a name in operations on the service. A token has an associated set of operations, specified (by a bitmask) when the token is requested; the token provides the right to perform these operations on the object to which it refers, bypassing any underlying protection mechanism. A token may have no access rights, in which case it serves simply as a name abbreviation that can be used in forming other names.

The token scheme can improve performance in two ways: 1) it eliminates the need for the service to check authorization on every operation; 2) it eliminates the need for SAM and the service to do name translation on each operation.

Service tokens may be discarded at any time by a service. This may be done either to limit table size, or to force periodic reauthorization in support of an "eventual revocation" policy. The client (or the client kernel) must remember the name and operation set, and be prepared to issue another token request. A token does not represent a "session"; two tokens representing the same name and having the same rights are interchangeable. Tokens are usable only during a crash-free period for both the client and the service instance.

4.2. DASH Global Architecture: Summary

The DASH global architecture defines a structure for global naming of permanent entities (hosts, owners and services), and for local naming of temporary entities (service tokens). The use of service tokens and distributed name caching are vertically integrated mechanisms in the sense that they involve both distributed and local architectures. The DASH global architecture is an open system in that it facilitates the addition of new services by

users. At a lower level, it is an open system in that 1) it provides an open framework for stream-oriented service protocols; 2) it supports inter-service naming; 3) authentication is factored from authorization.

The DASH global architecture compares favorably with those of other systems as a basis for VLDS. The V system [11] is a global system architecture intended for small distributed systems. It is not scalable; for example, it requires broadcast for service location. The Xerox Grapevine system [25] is a nameserver with poor scalability because of its nonhierarchical design. Other large-scale system architectures have used hierarchical naming for scalability and autonomy [9,28]. These are designed for limited purposes, such as host naming, and are not integrated with other types naming (such as the naming of files). Efforts at large-scale integration of existing centralized systems are described in [30] and [23]. These projects, for the most part, address restricted problems or develop solutions based on technology that will soon be outdated. In contrast, the DASH project is taking a unified approach to VLDS design, and is seeking solutions that will not be made obsolete by foreseeable technology advances.

Some distributed systems use *capabilities* for low-level naming and protection. Examples include Amoeba [18], Mach [22], and Eden [3]. Typically, a symbolic naming (directory) service is built on top of the capability mechanism. Arguments against the use of capabilities as a basis for identifying permanent objects are given in [4].

5. LOCAL SYSTEM ARCHITECTURE

By the *local system architecture* of an OS kernel we mean 1) the user-level abstractions provided for process control, system calls, and so on; 2) the dynamic (process and interrupt) structure of the kernel, and 3) the software engineering (programming) structure of the kernel. VLDS local system architecture has the following requirements:

- Support for user-level services: efficient IPC and control transfer between user-level processes, and support for user-level caching.
- Support for parallelism on shared-memory multiprocessors, both in the facilities provided to user programs and in the implementation of the kernel itself.
- Incorporation of modern software engineering ideas; the kernel must be maintainable, extensible, and portable across a range of architectures.

5.1. The DASH Local System Architecture

The DASH kernel is structured as a set of processes that share a single address space, communicating via message-passing and synchronized operations on shared data objects. The kernel uses multiple processes wherever possible. Communication protocols, RKOM servers, device drivers, system calls and other kernel functions execute as separate processes that can run in parallel on a multiprocessor. Work is done in processes that might be done in hardware or software interrupt handlers on other systems.

The kernel provides *user virtual address spaces*, each occupied by zero or more *user processes*. Each user address space has a set of *protected object references* to objects in the kernel.

5.1.1. Message-Passing

Message-passing (MP) is used for many purposes in DASH:

(1) Interprocess communication (IPC) between processes on a single host: user processes, communication protocols (which are implemented as processes), and other kernel-level processes.

(2) Control transfer between address spaces. For example, system calls (during which a process switches from user to kernel space and back) use MP.

(3) Allocation, buffering and queueing of data buffers: for example, network interface queues and virtual memory page pools.

(4) Implementation of other synchronization mechanisms such as timers, multiple wait, semaphores, and read/write locks.

Because of this wide range of uses, the MP system must provide many semantic features, described below. The MP system allows clients to use (and pay for) exactly the features they need.

The MP system consists of two major parts:

- *Message representation*: A *message* is a logical array of bytes, implemented by a data structure consisting of a *header* and a set of not necessarily contiguous *data areas*. The interface to messages is a set of operations for creating, manipulating and accessing messages. Message headers contain space for parameters to message-passing operations.
- *MP operations* (some of which are described below). This part is *extensible*; instead of having a single object type (port, mailbox, and so on) as the target of MP operations, there are several such types. It is simple to add new types as the kernel is developed. The various MP object types provide a variety of message-passing operation semantics. These operations have four binary degrees of freedom, yielding 16 logical combinations; all are possible and potentially useful, but only a subset are currently supported in DASH.

  Stream vs. request/reply: in *stream mode*, message flow is unidirectional, while in *request/reply mode*, message exchanges occur in synchronized pairs.

  Uniprocess vs. dual-process: a stream-mode message, or the request message in a request/reply operation, may be processed in the context of the sending process, or by separate process.

  Mode of sender: kernel processes invoke MP operations by making procedure calls. MP operations can also be invoked from user-level processes. The semantics are essentially the same as for kernel processes, but user-level MP operations are initiated via a trap instruction; the kernel trap handler completes the operation.

  Mode of receiver: receive processing may be done in a different mode than that of the sender. For example, DASH system calls are implemented as uniprocess MP operations that are initiated in user mode and processed in kernel mode.

An MP object supports either stream mode or request/reply mode operations. Each mode is represented by a base class whose virtual functions are the generic operations on MP objects of that mode. Derived classes implement these functions. Clients, in general, do not have to know the exact type of MP object, only its base class. This *object-oriented*

MP system was motivated by the following considerations:

- *System calls*: these are message-passing operations directed to *system call* MP objects. By default, the system call object is uniprocess, and system calls are executed by the calling process in kernel mode (no process switch is done). By substituting a dual-process system-call object, system calls can be redirected to other processes (e.g., for debugging purposes) transparently to the user process.
- The DASH network communication architecture allows stream protocols to be dynamically configured. Each protocol is a process, and communication between protocols is via stream-mode MP objects, but the ST layer allows network messages to be sent by procedure call. Hence, if the MP system had been designed differently, it would have been necessary for a protocol process to know whether it was connected directly to the subtransport layer, or to an intervening protocol.

Some MP objects may provide additional features:

- Messages may convey scheduling deadlines between processes.
- Certain MP objects (of both modes) serve as the means of accessing a "pool of servers". It is often desirable to have multiple server processes, so that multiple requests can be executed in parallel. The optimal number of servers may not be known in advance. Such MP objects may use a feature called *automatic receiver creation*. If a message is sent to such an MP object and there is no process waiting to receive it, a process will automatically be created.
- Dual-process stream MP objects act as buffers between producer and consumer processes. Operations on such objects can be subjected to *flow control*. Flow control may be based either on the number of queued messages, or the amount of data in the queue. It is also possible to use hysteresis for both the sender and the receiver. This can reduce the number of context switches, since a process can handle a batch of messages in one context switch.

5.1.2. Kernel Memory Management

The DASH kernel dynamically creates objects. Pointers to these objects may be distributed throughout the kernel, so it is not always safe to deallocate their memory. This creates a potential problem of unbounded memory usage. This problem is dealt with in two ways.

First, the kernel executes in a virtual address space, part of which is pageable. The constructor for each object class specifies whether memory is to be allocated from the pageable or nonpageable part. Objects that are cache entries (such as objects in the name service cache) are kept in the pageable part. Therefore the VM page replacement scheme (e.g., an LRU approximation) is inherited by all kernel caches.

Second, certain types of objects (such as ports) have the properties that 1) many references to the object can exist, and it is not feasible to keep track of them; 2) the object can be deleted at any time, and this must be detected on any future reference to the object. In DASH, these objects are allocated using a *pseudo-permanent object* facility. Memory blocks allocated for such objects are preceded by a *unique ID* field that is cleared when the object is freed. A memory block allocated for this purpose can be reused for other pseudo-permanent objects, but not for other purposes. A reference to a pseudo-

permanent object consists of a pointer to the memory block and a UID value; if this value does not match, then the object has been deleted.

Together, these two techniques eliminate the need for general-purpose garbage collection in the DASH kernel.

5.1.3. Kernel Software Engineering

The DASH kernel is being implemented in an object-oriented language, C++ [27]. The kernel is structured as a set of classes (abstract types), of which some have multiple dynamically-created instances. In keeping with the principles of object-oriented programming, the class/object structure encapsulates design decisions and machine dependencies. Our intent is to produce a maintainable, extensible and portable kernel. We view the DASH project as an important test case in the application of object-oriented techniques to OS kernel implementation.

5.2. The DASH Local Architecture: Summary

The DASH local architecture is *vertically integrated* in several ways:

- The dynamic structure of the kernel is a set of processes that communicate through a versatile message-passing facility. The same structure is available at the user level, and the system call interface uses message-passing.
- The static structure of the kernel is a set of objects, some of which are pseudo-permanent. The same structure is presented at the user level using protected object references, and is presented remotely via remote object references.

The DASH local architecture provides considerable "openness". The kernel process model and the object-oriented message-passing system simplify the task of experimenting with kernel parallelism. In addition, several mechanisms that are entangled in the kernel of other systems are moved to the user level in DASH:

- Process control and exception handling.
- Current directory or other context mechanisms.
- Transaction management.

The themes of the DASH local architecture are related to those of many existing operating systems. Other systems [8, 19] use message-passing in their kernel implementation for increased parallelism. Message-passing for exceptions, system calls and process control is used in V [11].

6. CONCLUSION

The DASH project at UC Berkeley is studying the design principles of future distributed systems. Based on technological trends, we predict the development of very large distributed systems (VLDS) based on high-performance wide-area networks and providing global access to a variety of data and computing resources. We have made the following general conclusions:

(1) The requirements of the performance and flexibility of low-level mechanisms (virtual memory, process control, kernel structure, naming, local IPC, and network communication) in a VLDS are not met by the low-level mechanisms of current distributed systems. New designs and experimentation are needed in these areas.

(2) The optimal low-level mechanisms for a VLDS often involve 1) the *vertical integration* of components at different system levels, and 2) an *open system* design in which the standardized portions of the system are minimal.

At this point (February 1988) much of the DASH design is in place, and the implementation of the kernel is proceeding. We plan on completing the basic design, completing uniprocessor and multiprocessor implementations of the kernel, and evaluating the basic design. Once this is done, the DASH system can serve as a testbed for research and development in many new and unexplored areas of distributed systems, especially those involving real-time communication and very large-scale replication and distributed processing.

There are also many areas for research within the design areas discussed in this report:

- In the IPC area, several issues involving RMS remain to be investigated. How can RMS be implemented on current networks and internetworks? What are appropriate stream transport protocols? What new types of applications does RMS make possible? How can multicast be implemented on top of RMS? Others problems involve the VM-based data movement system; these will be explored when DASH is ported to shared-memory multiprocessors.
- In the global architecture, the performance of the global naming system must be investigated. Other problems include the design of protection mechanisms in a VLDS, the design of highly replicated data servers, and the formal analysis of trust relationships in naming.
- In the local architecture, research problems involve 1) multiprocessor synchronization (both the mechanisms themselves, and their integration in programming languages), 2) process control for remote debugging, 3) process scheduling for multiprocessors, and 4) support for caching in user-level services.

7. ACKNOWLEDGEMENT

We would like to thank the following people for their contributions to the DASH project: Brian Bershad, G.D. Giuseppe Facchetti, Kevin Fall, G. Scott Graham, Ellen Nelson, P. Venkat Rangan, Bruno Sartirana, Shin-Yuan Tzou, Raj Vaswani, and Robert Wahbe.

REFERENCES

1. M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young, "Mach: A New Kernel Foundation for UNIX Development", *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, June 9-13, 1986, 81-92.
2. R. Agrawal, "Location Independent Remote Execution in NEST", *IEEE Trans. on Software Eng. 13*, 8 (August 1987), 905-912.
3. G. T. Almes, A. P. Black, E. Lazowska and J. Noe, "The Eden System: A Technical Review", *IEEE Transactions on Software Engineering 11*, 1 (January 1985), 43-59.
4. D. P. Anderson, D. Ferrari, P. V. Rangan and S. Tzou, "The DASH Project: Issues in the Design of Very Large Distributed Systems", Technical Report No. UCB/Computer Science Dpt. 87/338, Computer Science Division, EECS, UCB, Berkeley, CA, Jan. 1987.
5. D. P. Anderson, "A Software Architecture for Network Communication", Technical Report No. UCB/Computer Science Dpt. 87/386, Computer Science Division, EECS, UCB, Berkeley, CA, December 1987.
6. D. P. Anderson, D. Ferrari and P. V. Rangan, "Subtransport Level: The Right Place for End-to-End Security Mechanisms", Technical Report No. UCB/Computer Science Dpt. 87/346, Computer Science Division, EECS, UCB, Berkeley, CA, March 1987.
7. D. P. Anderson and P. V. Rangan, "A Basis for Secure Communication in Large Distributed Systems", *IEEE Symposium on Security and Privacy*, April 1987.
8. E. Basart, "The Ridge Operating System: High Performance through Message-Passing and Virtual Memory", *Proc. of the IEEE 1st International Conf. on Computer Workstations*, San Jose, California, Nov. 11-14, 1985, 134-143.
9. A. D. Birrell, B. W. Lampson, R. M. Needham and M. D. Schroeder, "A Global Authentication Service without Global Trust", *IEEE Symposium on Security and Privacy*, 1986.
10. D. R. Cheriton and W. Zwaenepoel, "The Distributed V Kernel and its Performance for Diskless Workstations", *Proc. of the 9th ACM Symp. on Operating System Prin.*, Bretton Woods, New Hampshire, Oct. 10-13, 1983, 128-140.
11. D. R. Cheriton, "The V Kernel: a Software Base for Distributed Systems", *IEEE Software 1*, 2 (April 1984), 19-43.
12. T. P. Dobry, A. M. Despain and Y. N. Patt, "Performance Studies of a Prolog Machine Architecture", *12th International Symposium on Computer Architecture*, 1985.
13. D. D. Gajski and J. Peir, "Essential Issues in Multiprocessor Systems", *IEEE Computer*, June, 1985, 9-28.

14. G. Gawrys, P. Marino, G. Ryva and H. Shulman, "ISDN: Integrated Network/Premises Solutions for Customer Needs", *IEEE Int. Comm. Conf.*, June 1986, 2-6.

15. M. Hill, "Design Decisions in SPUR", *IEEE Computer*, November 1986, 8-22.

16. W. D. Hillis, "The Connection Machine", *MIT Press, Cambridge, Mass.*, 1985.

17. M. Liu, D. Messerschmitt and D. Hodges, "An Approach to Fiber Optic Data/Voice/Video LAN", *IEEE INFOCOM 86*, 1986, 516-523.

18. S. Mullender and A. Tanenbaum, "Protection and Resource Control in Distributed Operating Systems", *Computer Networks 8*, 5,6 (1984), 421-432.

19. R. Olson, "Parallel Processing in a Message-Based Operating System", *IEEE Software*, July 1985, 39-49.

20. J. Postel, "Transmission Control Protocol", *DARPA Internet RFC 793*, September 1981.

21. D. M. Ritchie, "A Stream Input-Output System", *Bell System Tech. J. 63*, 8 (October 1984), 1897-1910.

22. R. D. Sansom, D. P. Julin and R. F. Rashid, "Extending a Capability Based System into a Network Environment", *1986 SIGCOMM Symposium*, , 265-274.

23. R. E. Schantz, R. H. Thomas and G. Bono, "The Architecture of the Cronus Distributed Operating System", *Proc. 6th Int. Conf. on Distributed Computing Systems*, May 1986, 250-259.

24. R. W. Scheifler and J. Gettys, "The X Window System", *ACM Transactions on Graphics 5*, 2 (April 1986), 79-109.

25. M. D. Schroeder, A. D. Birrell and R. M. Needham, "Experience with Grapevine: the Growth of a Distributed System", *ACM Transactions on Computer Systems 2*, 1 (February 1984), 3-23.

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

27. B. Stroustrup, "The C++ Programming Language", *Addison-Wesley*, 1986.

28. D. B. Terry, M. Painter, D. W. Riggle and S. Zhou, "The Berkeley Internet Domain Server", *Proceedings of the 1984 Summer USENIX Conference*, Salt Lake City, Utah, June 12-15, 1984, 23-31.

29. D. B. Terry, "Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments", Technical Report No. UCB/Computer Science Dpt. 85/228, Computer Science Division, U.C. Berkeley, March 1985.

30. T. Truscott, B. Warren and K. Moat, "A State-Wide UNIX Distributed Computing System", *Proceedings of the 1986 Summer USENIX Conference*, Atlanta, Georgia, June 9-13, 1986, 499-513.

31. J. Turner, "Design of a Broadcast Packet Network", *IEEE INFOCOM 86*, , 667-675.

32. S. Tzou, D. P. Anderson and G. S. Graham, "Efficient Local Data Movement in Shared-Memory Multiprocessor Systems", *Technical Report No. UCB/Computer*

Science Dpt. 87/385, Berkeley, CA, December 1987.

33. "The DASH Communication Architecture", UCB/Computer Science Dpt. Technical Report, in preparation, Feb. 1988.
34. "The DASH Local System Architecture", UCB/Computer Science Dpt. Technical Report, in preparation, Feb. 1988.
35. "The DASH Virtual Memory System", UCB/Computer Science Dpt. Technical Report, in preparation, Feb. 1988.