A Distributed Systems Architecture for the 1990's

by Butler W. Lampson, Michael D. Schroeder and Andrew D. Birrell

Sunday, December 17, 1989

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

A Distributed Systems Architecture for the 1990's

Butler W. Lampson, Michael D. Schroeder and Andrew D. Birrell

Most markets for computing are evolving towards distributed solutions. The system framework that accomodates distributed solutions most gracefully is likely to dominate in the 1990's.

A leadership distributed system includes the best of today's centralized systems, combining their coherence and function with the better cost/performance, growth, scale, geographic extent, availability, and reliability possible in distributed systems.

To build such a system requires a distributed systems architecture as the framework for a wide variety of products. The architecture specifies a set of standard services in a distributed system. Together these services make up an integrated system with global names, access, availability, security, and management, all working uniformly throughout the system.

This report summarizes a complicated subject in only ten pages (not counting the appendix). We made it as short as we could. Please read it all.

CONTENTS

1. WHY DISTRIBUTED SYSTEMS?

A distributed system is several computers doing something together. Compared to a centralized system, a distributed system can

- allow more widespread sharing of information and resources;
- provide more cost-effective computing by using cheap workstation computing cycles;
- be more available and more reliable;
- be more easily expanded and upgraded;
- cover bigger areas and serve larger numbers of users.

No wonder everybody wants a distributed system.

Customers with computer networks based on protocol families such as DECnet or TCP/IP already get many of the benefits of distributed systems. These networks are widespread today. Customers have a taste of distributed computing, and they like it. There's no turning back. Computing is going to be more and more distributed.

But today's networked systems are harder to use than their centralized predecessors. Centralized systems provide many functions and a lot of system-wide coherence: all the resources of the system can be accessed and managed in the same way from any part of it. Today, the functions on a single computer don't necessarily work over the network. For example, it may be impossible to let a colleague have access to a file if he is not registered locally. And many functions on the network don't work the same way they do locally, and work differently from computer to computer. For example, for a user of a Unix system (even when using NFS), reading a local file is not the same as reading a remote file from another system over the network. In VMS, you can copy a remote file, but you can't print it.

Functionality and coherence decrease as networked systems get larger. Today's large networked computer systems (and many small ones) are a set of independent computers interconnected by a network, not an integrated distributed system. Sharing among the computers is generally limited to mail transport, file transfer, and remote terminals. These sharing mechanisms can span many computers because of the underlying packet-switching network, but they aren't integrated into the usual way of doing business within each system. A few integrated distributed applications have been developed. But basically each system in the network is still a world of its own, with its own user registrations, file system configuration, and way of doing things. Customers exhibit a growing sense of frustration about the lack of function and coherence as their networked systems grow larger. No wonder no one is happy with the distributed system he's got.

The development of engineering workstations has made the problem worse, by increasing the rate at which computers are added to networks. Customers want workstations because they provide dramatically more cost-effective performance than time-sharing systems. Workstations also allow fine-grained expansion and upgrading. But having more computers forces more use of the poorly integrated remote functions. And from a management point of view each workstation is its own tiny time-sharing system: in large numbers they become a major headache.

The redundancy and independent failure obtained by having many interconnected computers can be exploited to increase reliability and availability. But not many systems have the software needed to duplicate functions and provide smooth fail-over. In most of today's distributed systems, a user doesn't see non-stop operation; instead, he may not be able to get his work done because of the failure of a computer he's never even heard of.

A leadership distributed system, then, is

- A heterogeneous set of hardware, software, and data *components*,

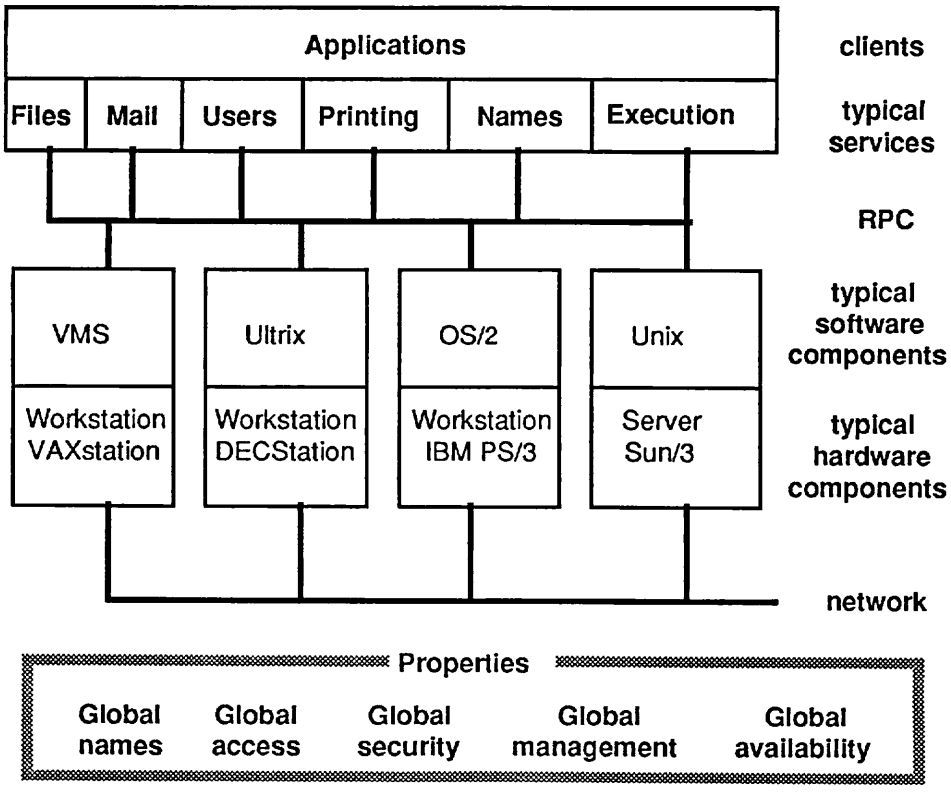

Figure 1: A leadership distributed system

- whose size and geographic extent can vary over a large range,
- connected by a network,
- providing a uniform set of *services* (user registration, time, files, records, printing, program execution, mail, terminals),
- with certain *global properties* (names, access, security, management and availability).

The coherence that makes it a system rather than a collection of machines is a result of uniform services and global properties. The services are available in the same way to every part of the system, and the properties allow every part of the system to be viewed in the same way. Let's see how close the state of the art is to this goal, and what we must do to reach it.

2. SOME EXISTING PRODUCTS

Current distributed system products offer either coherence or scale, but not both in the same system. They provide few fault-tolerant services and applications. Let's review the capabilities of a few current product distributed systems:

- **Inter-connected time-sharing systems:** Many of the currently available products are in reality collections of single- and multi-user timesharing systems using network technology to provide some inter-connection. This is true, for example, of systems composed of DECnet with VMS, Ultrix, etc., and of systems composed of TCP/IP with SunOS, NFS, etc. There are almost no fault tolerant services within this framework. There is little coherence in the way resources and information are named, accessed, or managed across the network. On the plus side, such systems admit a variety of computers and operating systems, are easily expandable, scale to very large numbers of computers, and can span the globe.
- **Tightly-coupled systems:** Other systems, such as VAXclusters and Apollo Domain offer distribution with a high degree of coherence. For example, a VAXcluster provides a common file system, batch queue, print queues, user registration, management, locking, etc. And the Domain system provides a system-wide shared virtual memory and file system However, there are not many fault-tolerant applications to run on this base. While these systems are incrementally expandable, the size limit is low and the geographic span is limited. For example, when upgrading the software in a VAXcluster, all cluster members must run the same version of VMS. And all cluster members must be Vaxes. Similar restrictions apply in a Domain system.

- **Fault Tolerant Systems:** Tandem is a leader in the use of local distribution to provide fault-tolerant applications. Their distributed systems are limited in other respects, but will certainly evolve towards the goals defined here.
- **IBM:** CICS (Customer Information Control System) is an extremely successful product offering remote invocation and distributed commit. In the future, IBM can be expected to build on its strengths in transaction processing and personal computers.

3. A LEADERSHIP DESIGN

The industry can and will satisfy customers' demands for increased function and coherence in distributed systems. The only question is which design will lead the way. The outlines of a 1990's leadership design are easy to see: combine the virtues of a centralized system with the virtues of distributed computing. Centralized systems offer function and coherence. Today's distributed systems offer interconnection, cost-effective computing, and growth. Tomorrow's distributed systems can provide all these—function, coherence, interconnection, cost-effective computing, and growth. In addition, they can offer new levels of availability and reliability.

4. A COHERENT DISTRIBUTED SYSTEM ARCHITECTURE

To make a distributed system coherent and highly functional, we need a standard way of doing things and a set of standard services that pervade all the

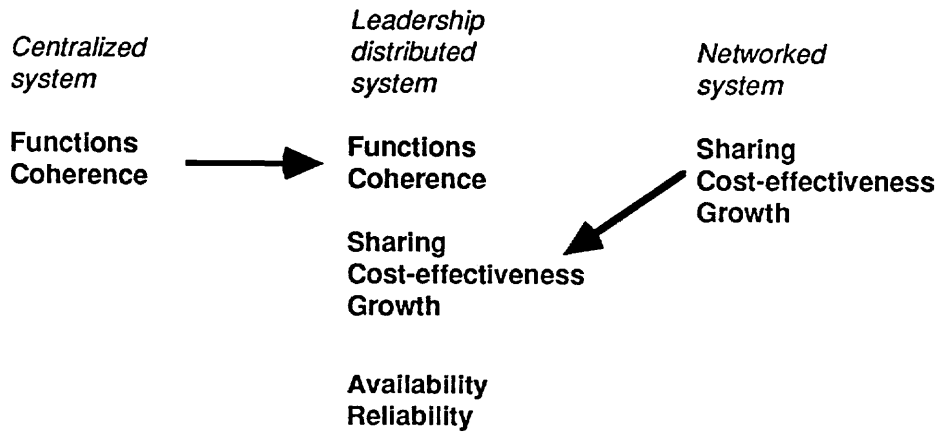

Figure 2: Getting to a leadership distributed system

computers of the system. To provide these standards we need an architecture, which we will call the Coherent Distributed System Architecture, CDSA for short. The architecture provides the homogeneity that makes the distributed system a *system*, rather than just a collection of computers.

But this homogeneity doesn't mean that all the components of the system must be the same. CDSA applies to a heterogeneous collection of computers running systems such as Unix, VMS, MS-DOS, and others. In short, all computers and operating systems can operate in this framework, even computers and systems from multiple vendors. The underlying network is a collection of Ethernets, new higher-speed LANs, bridges, routers, gateways, various types of long distance links, with connectivity provided by OSI transport and other protocols.

CDSA provides global names, global access, global security, global management, and global availability. "Global" means everywhere in the system. For example, Digital's internal engineering network, "EasyNet' currently has about 50,000 nodes and is growing rapidly. If it was converted to be a CDSA-based system—call it "BetterNet"—then "global" would include all 50,000 of these computers all over the world.

CDSA scales over a wide range of system sizes and extents, from small, local installations with a few computers in one building to world-wide systems such as BetterNet with 50,000 or more computers.

Let's examine what these pervasive properties mean:

- **Global names:** The same names work everywhere. Machines, users, files, distribution lists, access control groups, and services have full names that mean the same thing regardless of where in the system the names are used. For instance, Butler Lampson's user name might be something like `DEC.Eng.SRC.BWL` throughout BetterNet. He will operate under that name when using any computer on BetterNet. Global naming underlies the ability to share things.

- **Global access:** The same functions are usable everywhere with reasonable performance. If Butler sits down at a machine in Palo Alto, he can do everything there that he can do in Cambridge, with perhaps some small performance degradations. For instance, Butler could command the printing facilities on a computer in Palo Alto to print a file stored on his computer in Cambridge. Global access also includes the idea of data coherence. Suppose Butler is in Cambridge on the phone to Mike Schroeder in Palo Alto. Butler makes a change to a file and writes it. If they're both on BetterNet, Mike will automatically be able to read the new version as soon as it has been written. Neither Mike nor Butler needs to take any special action to make this possible.

- **Global security:** The same user authentication and access control work everywhere. For instance, Butler can authenticate himself to any computer on BetterNet; he can arrange for data transfer secure from eavesdropping and modification between any two BetterNet computers;

and assuming that the access control policy permits it, Butler can use exactly the same BetterNet mechanism to let the person next door and someone from another group read his files. All the facilities that require controlled access—logins, files, printers, management functions, etc.—use the same machinery to provide the control.

- **Global management:** The same person can manage components anywhere. Obviously one person won't manage all of BetterNet. But the system should not impose *a priori* constraints on which set of components a single person can manage. All of the components of the system provide a common interface to management tools. The tools allow a manager to perform the same action on large numbers of components at once. For instance, a single system manager could configure all the workstations in an organization without leaving his office.

- **Global availability:** The same services work even after some failures. System managers get to decide (and pay for) the level of replication for each service. As long as the failures don't exceed the redundancy provided, each service will go on working. For instance, a group might decide to duplicate its file servers but get by with one printer per floor. System-wide policy might dictate a higher level of replication for the underlying communication network. On BetterNet, mail won't need to fail between Palo Alto and Cambridge just because a machine goes down in Lafayette, Indiana.

The standard services defined by CDSA include the following:

- **Names:** Access to a replicated, distributed database of global names and associated values for machines, users, files, distribution lists, access control groups, and services. A name service is the key CDSA component for achieving global names, although most of the work involved in achieving that goal is making all the other components of the distributed system use the name service in a consistent way.

- **Remote procedure call:** A standard way to define and invoke service interfaces. Allows service instances to be local or remote.

- **User registrations:** Allows users to be registered and authenticated. Issues certificates permitting access to system resources and information. An architecture such as Kerberos addresses some of this area, although Kerberos does not meet all of the requirements for CDSA.

- **Time:** Consistent and accurate time globally.

- **Files:** A replicated, distributed, global file service. Each component machine of the distributed system can make available the files it stores locally through this standard interface. Information stored in the name

service together with file system clerk code running in client machines knits these various local files systems together into the coherent global file system. The file service specification should include standard presentations for the different VMS, Unix, etc. file types. For example, all implementations should support a standard view of any file as an array of bytes. There is presently no published architecture for a file service that meets CDSA's requirements.

- **Records:** An extension of the file service to provide access to records, either sequentially or via indexes, with record locking to allow concurrent reading and writing, and journalling to preserve integrity after a failure.

- **Printers:** Printing throughout the network of documents in standard formats such as Postscript and ANSI, including job control and scheduling.

- **Execution:** Running a program on any machine (or set of machines) in the network, subject to access and resource controls, and efficiently scheduling both interactive and batch jobs on the available machines, taking account of priorities, quotas, deadlines and failures. For many customers, the most cost-efficient, upgradable, available, reliable configurations include small numbers of medium- to large-sized cycle servers. The exact configuration and utilization of cycle servers (not to mention idle workstations that can be used for computing) fluctuates constantly, so users and applications need automatic help in picking the machines on which to run.

- **Mailboxes:** A computer message transport service, based on appropriate international standards. X.400 is probably adequate for defining this service.

- **Terminals:** Access to a windowing graphics terminal from a computation anywhere in the network. X-windows is a suitable architecture for this service.

- **Accounting:** System-wide collection of data on resource usage which can be used for billing.

CDSA includes a specification of the interface to each of these services. The interface defines the operations to be provided, the parameters of each, and the detailed semantics of each relative to a model of the state maintained by the service. The specification is normally represented as an RPC interface definition.

Each service can be provided in multiple implementations, but all must conform to the specified interfaces. CDSA also must specify how each service will provide the five pervasive properties: global names, global access, global security, global management, and global availability.

Our definition of a distributed system assumes a *single* set of interfaces for the standard services and global properties. For example, every component of the system can be named, accessed, and managed in the same way. Further, every component that provides or consumes a service, such as file storage or printing, does so through the same interface. There still may be several implementations of the interfaces for naming, management, files, etc., and this variety allows the system to be heterogeneous. In its interfaces, however, the system is homogeneous.

It is this homogeneity that makes it a *system* with predictable behavior rather than a collection of components that can communicate. If more than one interface exists for the same function, it is unavoidable that the function will work differently through the different interfaces. The system will consequently be more complicated and less reliable. Perhaps some components will not be able to use others at all because they have no interface in common. Certainly customers and salesmen will find it much more difficult to configure workable collections of components.

In reality, of course, there is no absolute distinction between a system and a collection of components. A system with more different ways of doing the same thing, whether it is naming, security, file access, or printing, is less coherent and dependable. On the other hand, it can still do a lot of useful work. The evils of heterogeneous interfaces can be mitigated by *gateways*, components that map from one interface to another. A gateway is often necessary when two systems evolve separately and later must be joined into one. An obvious example is the conjunction of IBM's System Network Architecture (SNA) and Digital's DECnet, and indeed there are already several DECnet/SNA gateways.

5. BENEFITS OF CDSA

A system conforming to CDSA will be of enormous benefit to customers:

- A system can contain a variety of hardware and software from a variety of vendors who each support CDSA..
- A system can be configured to have no single point of failure. The customer gets service nearly all the time, even when some parts of the system are broken.
- A system can grow by factors of thousands (or more), from a few computers to 100,000. It can grow a little at a time. Nothing needs to be discarded.
- Managers and users of that large collection of components can think of it as a single system, in which there are standard ways to do things everywhere. Resources can be shared—especially the data they store—even when the computers are physically separate, independently managed, and different internally.

Customers will view CDSA as different from an integrated system such as VAXclusters or Domain because it allows a bigger system and supports multiple machine architectures and multiple operating systems. They will view CDSA as different from a network system such as DECnet, or Sun's current offerings because it defines all the services of a complete system and deals effectively with security, management, and availability.

CDSA has significant internal benefits to the vendors as well. It provides a standard set of interfaces to which they can build a variety of cost-effective implementations over a long time periodt. It provides building blocks that allow product teams to concentrate more on added value rather than base mechanisms. It produces systems that are easier to configure, sell, and support.

APPENDIX: MODELS FOR CDSA

Defining a Coherent Distributed Systems Architecture is feasible now because experience and research have suggested a set of models for achieving global naming, access, security, management, and availability. For each of these pervasive properties, we describe the general approach that seems most promising. This Appendix goes into more technical detail than the body of the report; feel free to skip it if you are not interested.

A. Naming model

Every user and client program sees the entire system as the same tree of named objects with state. A global name is interpreted by following the named branches in this tree starting from the global root. Every node has a way to find a copy of the root of the global name tree.

For each object type there is some service, whose interface is defined by CDSA, that provides operations to create and delete objects of that type and to read and change their state.

The top part of the naming tree is provided by the CDSA name service. The objects near the root of the tree are implemented by the CDSA name service. A node in the naming tree, however, can be a *junction* between the name service and some other service, e.g. a file service. A junction object contains:

- a set of servers for the named object
- rules for choosing a server
- the service ID, e.g. CDSA File Service 2.3
- an object parameter, e.g. a volume identifier.

To look up a name through a junction, choose a server and call the service interface there with the name and the object parameter. The server looks up the rest of the name. The servers listed in a junction object are designated by global names. To call a service at a server the client must convert the server name to something more useful, like the network address of the server machine and information on which protocols to use in making the call. This conversion is done with another name lookup. A server object in the global name tree contains:

- a machine name
- a "protocol tower".

A final name lookup maps the (global) machine name into the network address that will be the destination for the actual RPC to the service.

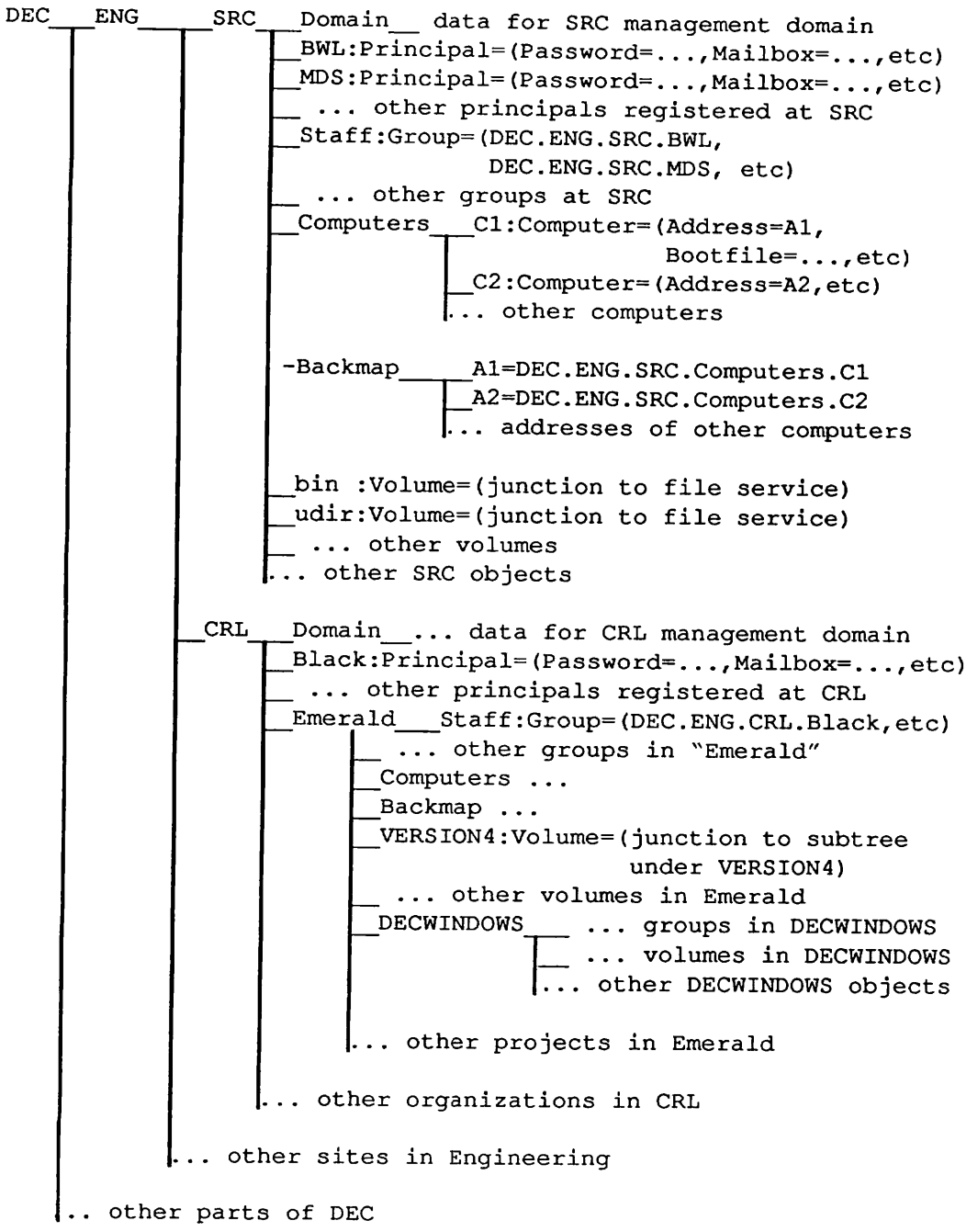

Figure 3: Top portion of a BetterNet global name space

Figure 3 gives an example of what the top parts of the global name space for BetterNet (the EasyNet replacement based on CDSA) might look like. An important part of the CDSA is designing the top levels of the global name space.

Consider some of the objects named in Figure 3. `DEC`, `DEC.ENG` , and `DEC.ENG.SRC` are directories implemented by the CDSA name service. `DEC.SRC.ENG.BWL` is a registered user of BetterNet, also an object implemented by the CDSA name service. The object `DEC.ENG.SRC.BWL` contains a suitably encrypted password, a set of mailbox sites, and other information that is associated with this system user. `DEC.ENG.SRC.staff` is a group of global names. Group objects are provided by the CDSA name service to implement things like distribution lists, access control lists, and sets of servers.

`DEC.ENG.SRC.bin` is a file system volume. Note that this object is a junction to the CDSA file service. Figure 3 does not show the content of this junction object, but it contains a group naming the set of servers implementing this file volume and rules for choosing which one to use, e.g., first that responds. To look up the name `DEC.ENG.SRC.bin.ls`, for example, the operating system on a client machine traverses the path `DEC.ENG.SRC.bin` using the name service. The result at that point is the content of the junction object, which then allows the client to contact a suitable file server to complete the lookup.

B. Access model

Global access means that a program can run anywhere in a CDSA-based distributed system (on a compatible computer and operating system) and get the same result, although the performance may vary depending on the machine chosen. Thus, a program can be executed on a fast cycle server in the machine room while still using files from the user's personal file system directory on another machine. Thus, a VMS printing program can print a file that is stored in a different machine or cluster.

Achieving global access requires allowing all elements of the computing environment of a program to be remote from the computer where the program is executing. All services and objects required for a program to run need to be available to a program executing anywhere in the distributed system. For a particular user, "anywhere" includes at least:

- on the user's own workstation;
- on public workstations or compute servers in the user's management domain;
- on public workstations in another domain on the user's LAN;
- on public workstations across a low-bandwidth WAN.

The first three of these ought to have uniformly good performance. Some performance degradation is probably inevitable in the fourth case, but attention should be paid to making the degradation small.

In CDSA, global naming and standard services exported via a uniform RPC mechanism provide the keys to achieving global access. All CDSA services accept global names for the objects on which they operate. All CDSA services are available to remote clients. Thus, any object whose global name is known can be accessed remotely. In addition, we must arrange that programs access their environments only by using the global names of objects. This last step will require a thorough examination of the computing environment provided by each existing operating system to identify all the ways in which programs can access their environment. For each way identified a mechanism must be designed to provide the global name of the desired object. For example, in Unix systems that operate under CDSA, the identities of the file system root directory, working directory, and /tmp directory of a process must be specified by global names. Altering VMS, Unix, and other operating systems to accept global names everywhere will be a major undertaking, and is not likely to happen in one release. As a result, we must expect incremental achievement of the global access goal.

Another aspect of global access is making sure CDSA services specify operation semantics that are independent of client location. For example, any service that allows read/write sharing of object state between clients must provide a data coherence model that is insensitive to client and server location. Depending on the nature of the service, it is possible to trade off performance, availability, scale, and coherence.

In Digital's DNS name service, for example, the choice is made to provide performance, availability, and scale at the expense of coherence. A client update to the name service database can be made by contacting any server. After the client operation has completed, the server propagates the update to object replicas at other servers. Until propagation completes, different clients can read different values for the object. But this lack of coherence increases performance by limiting the client's wait for update completion; increases availability by allowing a client to perform an update when just one server is accessible; and increases scale by making propagation latency not part of the visible latency of the client update operation. For the objects that the DNS name server will store, this lack of coherence is deemed acceptable. The data coherence model for the name service carefully describes the loose coherence invariants that programmers can depend upon, thereby meeting the requirement of a coherence model that is insensitive to client and server location.

On the other hand, the CDSA file service probably needs to provide consistent write sharing, at some cost in performance, scale, and availability. Many programs and users are accustomed to using the file system as a communication channel between programs. Butler Lampson may store a document in a file from his Cambridge workstation and then telephone Mike Schroeder in Palo Alto to say the new version of the document is ready. Mike will be annoyed if then fetching that file into his Palo Alto workstation produces the old version of the document. File read/write coherence is also important among elements of a distributed computation running, say, on multiple computers on

the same LAN. Most existing distributed file systems that use caching to gain performance do not have data coherence, e.g. Sun's NFS.

C. Security model

Security is based on three notions:

- Authentication: for every request to do an operation, the name of the user or computer system which is the source of the request is known reliably. We call the source of a request a "principal".
- Access control: for every resource (computer, printer, file, database, etc.) and every operation on that resource (read, write, delete, etc.), it's possible to specify the names of the principals allowed to do that operation on that resource. Every request for an operation is checked to ensure that its principal is allowed to do that operation.
- Auditing: every access to a resource can be logged if desired, as can the evidence used to authenticate every request. If trouble comes up, there is a record of exactly what happened.

To authenticate a request as coming from a particular principal, the system must determine that the principal originated the request, and that it was not modified on the way to its destination. We do the latter by establishing a "secure channel" between the system that originates the request and the one that carries it out. A secure channel might be a physically protected piece of wire, but practical security in a distributed system requires encryption to secure the communication channels. The encryption must not slow down communication, since in general it's too hard to be sure that a particular message doesn't need to be encrypted. So the security architecture includes methods of doing encryption on the fly, as data flows from the network into a computer.

To determine who originated a request, it's necessary to know who is on the other end of the secure channel. If the channel is a wire, the system manager might determine this, but usually it's done by having the principal at the other end demonstrate that it knows some secret (such as a password), and then finding out in a reliable way the name of the principal that knows that secret. CDSA's security architecture specifies how to do both these things. It's best if you can show that you know the secret without giving it away, since otherwise the system you authenticated to can impersonate you. Passwords don't have this property, but it can be done using public-key encryption.

It's desirable to authenticate a user by his possession of a device which knows his secret and can demonstrate this by public-key encryption. Such a device is called a "smart card". An inferior alternative is for the user to type his password to a trusted agent. To authenticate a computer system, we need to be sure that it has been properly loaded with a good operating system image; CDSA must specify methods to ensure this.

Security depends on naming, since access control identifies the principals that are allowed access by name. Practical security also depends on being able to have groups of principals (e.g., the Executive Committee, or the system administrators for the STAR cluster). Both these facilities are provided by DNS. To ensure that the names and groups are defined reliably, digital signatures are used to certify information in DNS; the signatures are generated by a special "certification authority" which is engineered for high reliability and kept off-line, perhaps in a safe, when its services are not needed. Authentication depends only on the smallest sub-tree of the full DNS naming tree that includes both the requesting principal and the resource; certification authorities that are more remote are assumed to be less trusted, and cannot forge an authentication.

D. Management model

System management is adjustment of system state by a human manager. Management is needed when satisfactory algorithmic adjustments cannot be provided; when human judgement is required. The problem in a large-scale distributed system is to provide each system manager with the means to monitor and adjust a fairly large collection of geographically distributed components.

The CDSA management model is based on the concept of domains. Every component in a distributed system is assigned to a domain. (A component is a piece of equipment or a piece of management-relevant object state.) Each domain has a responsible system manager. To keep things simple, domains are disjoint. Ideally a domain would not depend on any other domains for its correct operation.

Customers will require guidelines for organizing their systems into effective management domains. Some criteria for defining a domain might be:

- components used by a group of people with common goals;
- components that a group of users expects to find working;
- largest pile of components under one system manager;
- pile of components that isn't too big.

A user ought to have no trouble figuring out which system manager to complain to when something isn't working satisfactorily. That manager ought to have some organizational incentive to fix the user's problem.

CDSA requires that each component define and export a management interface, using RPC if possible. Each component is managed via RPC calls to this interface from interactive tools run by human managers. Some requirements for the management interface of a component are:

- Remote access: The management interface provides remote access to all system functions. Local error logs are maintained that can be read from the management interface. A secure channel is provided from

management tools to the interface. No running around by the manager is required.

- Program interface: The management interface of a component is designed to be driven by a program, not a person. Actual invocation of management functions is by RPC calls from management tools. This allows a manager to do a lot with a little typing. A good management interface provides end-to-end checks to verify successful completion of a series of complex actions and provides operations that are independent of initial component state to make it easier to achieve the desired final state.

- Relevance: The management interface should operate only on management- relevant state. A manager shouldn't have to manage everything. In places where the flexibility is useful rather than just confusing, the management interface should permit decentralized management by individual users.

- Uniformity: Different kinds of components should strive for uniformity in their management interfaces. This allows a single manager to control a larger number of kinds of components.

The management interfaces and tools make it practical for one person to manage large domains. An interactive management tool can invoke the management interfaces of all components in a domain. It provides suitable ways to display and correlate the data, and to change the management-relevant state of components. Management tools are capable of making the same state change in a large set of similar components in a domain via iterative calls. To provide the flexibility to invent new management operations, some management tools support the construction of programs that call the management interfaces of domain components.

E. Availability Model

To achieve high availability of a service there must be multiple servers for that service. If these servers are structured to fail independently, then any desired degree of availability can be achieved by adjusting the degree of replication.

Recall from the naming model discussion that the object that represents a service includes a set of servers and some rules for choosing one. If the chosen server fails, then the client can fail-over to a backup server and repeat the operation. The client assumes failure if a response to an operation does not occur within the timeout period. The timeout should be as short as possible so that the latency of an operation that fails-over is comparable to the usual latency.

To achieve transparent fail-over from the point of view of client programs, a service is structured using a clerk module in the client computer. With a clerk, the client program makes a local call to the clerk for a service operation. The clerk looks up the service name and chooses a server. The clerk then makes the RPC

call to the chosen server to perform the operation. Timeouts are detected by the clerk, which responds by failing over to another server. The fail-over is transparent to the client program.

As well as implementing server selection and fail-over, a clerk can provide caching and write behind to improve performance, and can aggregate the results of multiple operations to servers. As simple examples of caching, a name service clerk might remember the results of recently looked up names and maintain open connections to frequently used name servers. Write-behind allows a clerk to batch several updates as a single server operation which can be done asynchronously, thus reducing the latency of operations at the clerk interface.

As an example of how a clerk masks the existence of multiple servers, consider the actions involved in listing the contents of `DEC.ENG.SRC.udir.BWL.Mail.inbox` a CDSA file system directory. (Assume no caching is being used.) The client program presents the entire path name to the file service clerk. The clerk locates a name server that stores the root directory and presents the complete name. That server may store the directories `DEC` and `DEC.ENG`. The directory entry for `DEC.ENG.SRC` will indicate another set of servers. So the first lookup operation will return the new server set and the remaining unresolved path name. The clerk will then contact a server in the new set and present the unresolved path `SRC.udir.BWL.Mail.inbox`. This server discovers that `SRC.udir` is a junction to a file system volume, so returns the junction information and the unresolved path name `udir.BWL.Mail.inbox`. Finally, the clerk uses the junction information to contact a file server, which in this example actually stores the target directory and responds with the directory contents. What looks like a single operation to the client program actually involves RPCs to three different servers by the clerk.

For servers that maintain state that can be write-shared among multiple clients, providing high availability is complex. An example is a replicated file service such as needed for CDSA. The essential difficulties are arranging that no writes get lost during fail-over from one server to another, and that a server that has been down can recover the current state when it is restarted. Combining these requirements with caching and write-behind to obtain good performance, without sacrificing consistent sharing, can make implementing a highly available service quite challenging.

ACKNOWLEDGEMENT

Thank-you to Mary-Claire van Leunen for helping us with the writing.