DCE: An Environment for Secure Client/Server Computing

The Open Software Foundation's Distributed Computing Environment provides an infrastructure for developing and executing secure client/server applications that are portable and interoperable over a wide range of computers and networks.

by Michael M. Kong

The Distributed Computing Environment (DCE) is a suite of software that enables networked computers to share data and services efficiently and securely. The initial specification and development of DCE took place in 1989 under the aegis of the Open Software Foundation (OSF) through the OSF RFT (request for technology) process. Several companies in the computer industry, including HP, contributed technologies to DCE. HP has since released several versions of DCE as a product for HP-UX* systems, with added enhancements particularly in tools for administration and application development. HP remains active in the development of future OSF DCE releases.

The major technologies in DCE include:

- Threads. A library of routines that enable several paths of execution to exist concurrently within a single process.
- Remote Procedure Call (RPC). A facility that extends the procedure call paradigm to a distributed environment by enabling the invocation of a routine to occur on one host and the execution of the routine to occur on another.
- Security. A set of services for authentication (to verify user identity), authorization (to control user access to data and services), and account management. DCE security services are described in the article on page 41.
- Cell Directory Service (CDS). A service that maintains a database of objects in a DCE cell and maps their names (which are readable by human users) to their identifiers and locations (which are used by programs to access the objects). CDS is described in the article on page 23.
- Global Directory Service (GDS). A service that maintains a database of objects that may exist anywhere in the world and enables DCE programs to access objects outside a cell. GDS is also described in the article on page 23.
- Distributed Time Service (DTS). A service that synchronizes clocks on DCE hosts with each other and, optionally, with an external clock.
- Distributed File Service (DFS). A service that allows DCE hosts to access each other's files via a consistent global file naming hierarchy in the DCE namespace.

The HP DCE product adds several features to the OSF DCE offering, including:

- An integrated login facility that enables HP-UX login programs to perform DCE authentication for users. This feature is described in the article on page 28.
- A DCE cell configuration utility integrated with the HP-UX system administration manager (SAM).
- An object-oriented DCE (HP OODCE) programming environment that eases DCE application development for C++ programmers. HP OODCE is described in the article on page 55.
- Integration of DCE application development tools with the SoftBench product and extensions to these tools that support tracing and logging of distributed application activity.

This article describes the DCE client/server model, introduces DCE cells, and provides an overview of four technologies in DCE: threads, RPC (remote procedure calls), DTS (distributed time service), and DFS (distributed file service). Articles elsewhere in this issue describe DCE security, the DCE directory services, and other aspects of the HP DCE product. Unless otherwise specified, these articles describe version 1.4 of HP DCE/9000 as released for Version 10.10 of the HP-UX operating system.

The Client/Server Model

DCE applications and the various components of DCE interact according to a client/server model. Functionality is organized into discrete services; clients are users of services and servers are providers of services. A client program issues requests for services and a server program acts on and responds to those requests. A program may play both client and server roles at once by using one service while it provides another. For example, in a distributed application that relies on secure communication, both the client and server sides of the application also act as clients of DCE security services. The client/server model insulates the users of a service from the details of how the service is implemented, allowing the server implementation to be extended, relocated, or replicated without perturbing existing clients.

To make the service abstraction work in practice, clients and servers must agree on how they will interact. They agree on what requests the client can make of the server, and for each request, what data will flow between them. In DCE, these aspects of a service are described in a definition of the client/server interface written in the RPC Interface Definition Language (IDL). The DCE application development software ensures that client and server programs will adhere to the interface definition. Given an RPC interface definition for a service, an application developer can build and execute clients and servers on different DCE implementations, and the resulting programs will interoperate correctly.

In addition to RPC interfaces for distributed services, DCE defines application program interfaces (APIs) that applications invoke when they wish to use DCE services. In the example of the secure application mentioned above, the application client and the application server will both invoke DCE security routines provided by the DCE run-time library. The library will interact as necessary with the DCE security server on behalf of the application program. The existence of standard APIs for DCE services ensures the portability of applications across all DCE implementations.

DCE Cells

DCE services are deployed in administrative units called cells. A cell can encompass one host or many thousands of hosts in a single local network or in an internetwork spanning continents. The grouping of hosts into a cell does not necessarily follow physical network topology (though network performance characteristics may make some groupings more practical than others). Rather, a cell is usually defined according to administrative boundaries. A cell contains a single security database and a single Cell Directory Service (CDS) namespace, so all users and applications within a cell are subject to the same administrative authority, and resources are more easily shared within the cell than between cells.

Fig. 1 shows a relatively simple DCE cell containing servers and clients. The minimal set of services in a cell consists of a security server, a CDS server, and some means of synchronizing time among the hosts. In this cell, the security and CDS databases are replicated for increased performance and reliability, so there are two security servers and two CDS servers. The DCE time service, DTS, is used to synchronize clocks throughout the cell with an external time source. Other DCE services such as DFS and GDS (Global Directory Service) need not be configured in a minimal DCE cell but can be added at any time. A DCE-based application is installed in the cell in Fig. 1, with an application server running on an HP 9000 Series 800 server and application clients running on PCs and workstations. Finally, each host in the cell has a DCE run-time library and runs DCE client daemons.

Threads

In a distributed environment the need often arises for one program to communicate concurrently with several others. For example, a server program may handle requests from many clients. The DCE threads facility provides the means to create concurrent threads of execution within a process and hence eases the design and enhances the performance of distributed applications. The threads facility is not itself distributed, but virtually all distributed services in DCE rely on threads, as do most DCE-based applications.

POSIX (Portable Operating System Interface)[1] has defined an industry-standard programming interface for writing multithreaded applications: the POSIX 1003.4a specification. DCE threads is a user-space implementation of Draft 4 of this specification.

One way to introduce the notion of a thread is to describe an ordinary singled-threaded process and contrast this with a multithreaded process. A process is a running instance of a program. When a process starts, the text of the program is loaded into the address space of the process and then instructions in the program text are executed until the process terminates. The instructions that are executed can be thought of as a path or thread of execution through the address space of the process. An ordinary process can thus be considered to be single-threaded.

A threads facility allows several threads of execution to exist within one process. An initial thread exists when a process starts, and this thread can create additional threads, making

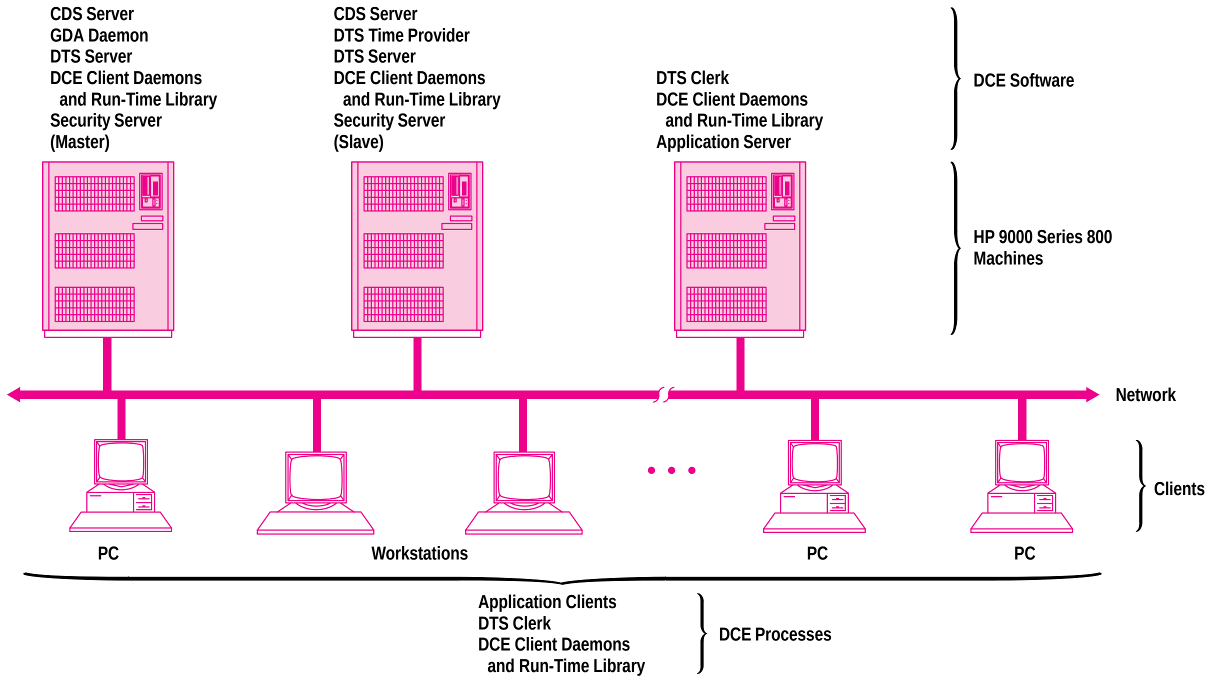

Fig. 1. A single DCE cell containing security, CDS, time, and application servers and application clients running on PCs and workstations.

the process multithreaded. Each thread executes independently and has its own stack. However, the threads in a process share most process resources, such as user and group identifiers, working directories, controlling terminals, file descriptors, global variables, and memory allocated on the heap.

Resource sharing and concurrent execution can lead to several performance benefits for multithreaded programs:

- Threads can be created, synchronized, and terminated much more efficiently than processes.
- If one thread blocks, waiting for I/O or for some resource, other threads can continue to execute.
- A server program can exhibit better responsiveness to clients by dedicating a separate thread to each client request. The server can accept a new request even while it is still executing older requests.
- On a multiprocessor computer, several threads within a process can run in parallel on several processors.

Of course, the execution of threads in a process can be truly concurrent only on a computer that has multiple processors and has a threads implementation that can take advantage of multiprocessing (even then, concurrency is limited by the number of processors). In reality, the threads in a process take turns executing according to a scheduling policy and a scheduling priority that are assigned to each thread. Depending on the policy that governs a thread, the thread will run until it blocks, until it consumes a time slice that was allocated to it, or until it voluntarily yields the processor. A context switch then occurs, and the next thread to execute is chosen from a queue of threads that are ready to run, based on their priorities.

Threads programmers can use condition variables to synchronize threads so that a thread will run only after a specified condition has been satisfied. A thread can wait on a condition variable either for a specified time to elapse or for another thread to signal that variable. The waiting thread does not reenter the ready queue until the condition is satisfied.

Because threads run concurrently and share process resources, programmers must protect regions of code that access shared resources. For example, if a context switch occurs in code that manipulates global variables, one thread may have undesired side effects on another thread. The threads API allows programmers to use mutual exclusion (mutex) locks to prevent such effects. Only one thread can hold a given mutex lock at any time, and any other thread that attempts to take the lock will block until the lock is released, so only the thread that holds the lock can execute the critical region of code.

Like global data, static data can be a conduit for side effects between threads when a context switch occurs, and this imposes another constraint on code that executes in multithreaded processes. Routines that can be called by multiple threads must not return pointers to static data.

The requirements mentioned above for code in multithreaded programs apply not only to DCE executables and DCE application programs, but also to any libraries used by those programs. A library is considered thread safe to the extent that it behaves correctly when used by a multithreaded program. The HP-UX operating system defines several levels of thread safeness for libraries. The HP-UX C library, for instance, can safely be called by several threads in one program, whereas some other libraries can be called by only one thread per program.

A kernel-space implementation of the final POSIX threads specification may ultimately replace the user-space implementation of Draft 4 that is currently supplied with HP DCE. Kernel threads would make true concurrency possible on multiprocessor computers and probably improve performance on uniprocessor machines as well.

Remote Procedure Call

The remote procedure call (RPC) facility is the basis for all DCE client/server communications and therefore is fundamental to the distribution of services in DCE applications and in DCE itself.

The RPC mechanism enables a procedure invoked by one process (the client) to be executed, possibly on a remote host, by another process (the server). The client and server hosts need not have the same operating system or the same hardware architecture. However, they do need to be able to reach each other via a transport protocol that is supported by the DCE implementations on both hosts.

DCE RPC conforms to a set of specifications collectively known as the Network Computing Architecture (NCA). The NCA specifications define the protocols that govern the interaction of clients and servers, the packet format in which RPC data is transmitted over the network, and the Interface Definition Language (IDL) that is used to specify RPC interfaces. DCE RPC is based on Version 2 of NCA. Version 1 of NCA was a set of architecture specifications for another remote procedure call facility, the Network Computing System (NCS), which has been in use on the HP-UX operating system and other platforms since the late 1980s. DCE RPC evolved from NCS, supports the interoperation of NCS and DCE applications, and offers features that assist in the conversion of applications from NCS to DCE.

NCA defines two RPC protocols, one for use over connection-based transports (called NCA CN RPC) and one for use over datagram-based transports (NCA DG RPC). The connection-based protocol relies on the underlying transport to manage connections between clients and servers, whereas the datagram-based protocol assumes an unreliable transport and performs its own connection management. A DCE implementation can support each of these protocols over several transports. HP DCE currently supports NCA connection-based RPC over TCP/IP and NCA datagram-based RPC over UDP/IP. The NCA protocols ensure that remote procedure call semantics are not affected by the underlying transport used. This characteristic of NCA, sometimes referred to as transport independence, is essential for the portability and interoperability of DCE and DCE applications over many types of networks and computers.

How RPC Applications Work. To understand how an RPC application works, first imagine an ordinary nondistributed program consisting of a main module, which performs various initialization tasks and handles user interaction, and a second module, which does the real work of the application such as interacting with a database. The main module can be thought of as a client of the services implemented and provided by the database module. In DCE terminology, a module that implements a service is called a manager, and the set of manager routines that the client calls constitutes the interface between client and manager.

Fig. 2a illustrates this simple program and a representation of the interface between the client and the manager pieces. Note that the modularization of this program demands only that the client and manager pieces adhere to the declared signature (calling syntax) of each routine in the interface. This implies that the manager module could be replaced by any other module containing routines that have the same names, return the same data type, and pass the same arguments. In an ordinary C application, routine signatures are typically declared in header files that get included in other modules.

Now imagine that this application is to be distributed so that the database management code executes on a minicomputer and the user interface code executes on a graphical workstation. The first step in building a DCE RPC application is to write an IDL interface definition. An interface definition specifies the UUID (universal unique identifier) and version of the interface, declares the signatures of the operations (routines) in the interface, and declares data types used by those operations (Fig. 2b). The declarations of types and operations in an IDL file resemble those in a C header file, but an IDL file contains additional information required to make the operations callable via RPC. For example, the operation declarations in an IDL file are embellished with attributes that specify explicitly whether the routine's arguments are inputs or outputs, so that when the routine is called, arguments pass over the network only in the direction needed.

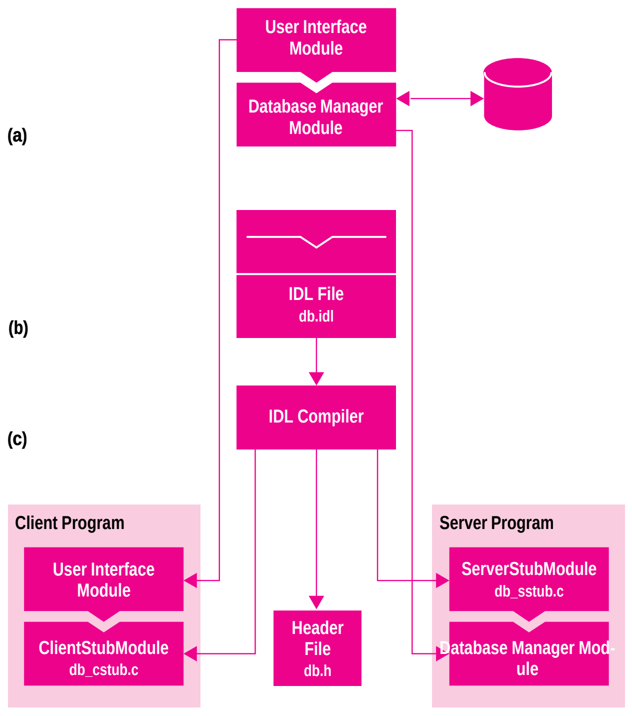

Fig. 2. The process of creating an RPC application. (a) Original application showing the part that will run on the client and the part that will run on a server. (b) Creating an IDL file. (c) Compiling the IDL file to create a header file and client/server stubs.

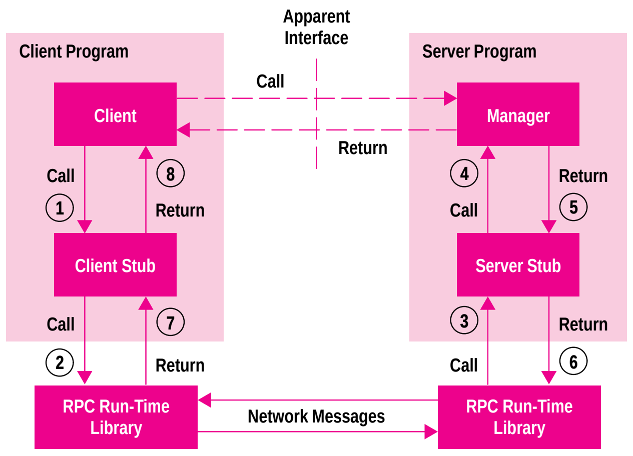

Fig. 3. Flow of events when a client program calls db_lookup on the server.

The next step in building a distributed application is to compile the interface definition with the DCE IDL compiler (Fig. 2c). The IDL compiler takes the IDL file as input and emits three C source files as output: a client stub module, a server stub module, and a header file. The IDL compiler derives the names of the emitted files from the name of the IDL file.

The client stub presents to the application client module the same interface that the manager module did in the local case. For example, if the manager module contained a routine called db_lookup, so will the client stub. Likewise, the server stub presents to the manager module the same interface that the application client module did. Continuing the example, the server stub calls the db_lookup routine in the manager just as the client did in the local case. The header file contains the declarations needed to compile and link the client and server programs.

The final step in building the application is to link these developer-written and IDL-compiler-generated modules into two programs: a client program consisting of the old client module and the client stub and a server program made up of the old manager module and the server stub. (This description is rather simplified. In reality, a number of DCE library APIs are typically invoked by application code in both the client and server programs.) Both programs are dynamically linked with the DCE shared library, which must be present as part of the DCE run-time environment on the client and server hosts.

Fig. 3 describes the flow of events that occur when the client program calls db_lookup. The call to db_lookup is resolved in the client stub module ①. The db_lookup routine in the client stub marshalls the operation's input parameters into a request buffer and then invokes routines in the DCE library to send the request to the server host ②. On both the sending and receiving sides, RPC code in the DCE library deals as necessary with any issues involving the underlying transport, such as fragmentation and window sizes. When the server program receives the request, DCE library code calls the db_lookup routine in the server stub module ③, which unmarshalls the input parameters and passes them to the actual implementation of db_lookup in the application manager module ④.

When the manager routine returns ⑤, the server stub marshalls the operation's output parameters and a return value into a response buffer and invokes DCE library routines to send the response to the client ⑥. Library code on the client side receives the response and returns control to the client stub db_lookup routine ⑦, which finally unmarshalls the outputs and returns to the main client module ⑧.

RPC Protocols. DCE RPC clients and servers communicate over a network by exchanging messages, such as the request and response messages described in Fig. 3. Each message, called a protocol data unit (PDU), consists of up to three parts:

- A header, which contains RPC protocol control information
- An optional body, which contains data
- An optional authentication verifier, which contains data for use by an authentication protocol.

The PDU itself is of course encapsulated by control information specific to the transport and network underlying a remote procedure call. For example, when a datagram-based RPC PDU is transmitted as a UDP/IP packet, it is preceded by UDP/IP header information.

A few examples of information that might be carried in the header of a DCE RPC PDU include:

- The version of the DCE RPC protocol in use
- The PDU type (Both connection-based RPC and datagram-based RPC define request and response PDU types. In addition, each RPC protocol defines several PDU types specific to the protocol. For example, because datagram-based RPC implements its own connection management, it defines PDU types for pings and acknowledgments.)
- The UUID that identifies the interface being used
- The operation number that identifies the operation being called within that interface
- The UUID that identifies the object on which the operation is to be performed
- A label that identifies the data representation format in which the PDU header and body data are encoded
- The length of the PDU body.

Many PDU types serve only to convey protocol control information between a client and server and hence have no body. Request and response PDUs, of course, do have bodies containing the input and output parameters of the remote procedure call. These parameters are encoded according to a transfer syntax identified by the data representation format label in the header. DCE RPC currently specifies only one transfer syntax, the network data representation (NDR) syntax.

NDR defines the representation of each IDL data type in a byte stream. For scalar types like integers and floating-point numbers, NDR addresses issues such as byte order and floating-point format. For constructed types like arrays, structures, and unions, NDR sets rules for flattening data into a byte stream. Thus, the set of input and output values in every remote procedure call has a byte stream representation determined by NDR syntax. The byte stream is passed between client and server as the body in one or more request and response PDUs. Table I lists the data types supported by RPC.

Some scalar data types have several supported formats in NDR. Integers, for example, may be in either big-endian (most significant byte first) or little-endian (least significant byte first) format. For these primitive types, the format that governs a particular PDU is indicated as part of the data representation format label in the PDU header. On any given hardware architecture, the DCE library will send outgoing data in the representations native to that architecture. If the receiving host has different native representations, its DCE library will convert incoming data (for example, by swapping bytes in integers) as necessary. DCE RPC thus has what may be called a multicanonical approach to data representation. This approach tends to minimize data conversion, and in particular, two hosts of the same architecture can usually communicate without ever converting data. By contrast, if a data representation scheme dictates a single canonical format for each scalar type, and the client and server hosts share a common format other than the canonical one, data will be converted both when sent and when received.

The third part of a DCE RPC PDU, the authentication verifier, is present only for authenticated remote procedure calls. It contains data whose format and content depend on the authentication protocol being used. Use of the authentication verifier is explained further in the description of authenticated RPC below.

Client/Server Binding. A key question in the design and implementation of a DCE RPC application is, how will the client locate an appropriate server? When making a remote procedure call, a client program must specify to its DCE run-time library the location of a server that can perform the requested operation. The server location incorporates an RPC protocol sequence (the combination of NCA protocol and network protocol), a network address or host name, and an endpoint (for the IP protocols, the endpoint is simply a port number). This information is encapsulated in a structure called a binding. A binding may also include the UUID of the object to be operated on, if any, and authentication and authorization information, if the call is to be authenticated.

The RPC API supports a range of techniques for obtaining and manipulating bindings. Most applications either construct a textual representation of a binding (called a string binding) from information supplied by the user or obtain a binding from a name service.

A string binding represents in a textual format the object UUID and server location portions of a binding. For example, in the string binding:

```
f858c02c-e42b-11ce-a344-080009357989@ncadg_ip_udp:192.18.59.131[2001]
```

the object UUID appears in the standard string format for UUIDs, the ncadg_ip_udp protocol sequence specifies the NCA DG RPC protocol over UDP/IP, an Internet address identifies the server host, and a port number specifies the endpoint on which the server is listening. (The object UUID and the endpoint are optional.)

Table I
Data Types Supported in RPC

Primitive Data Types	
Integers	
Floating-point numbers	
Characters	
boolean†	
byte†	A type usually used in arrays or structures to transmit opaque data. Data of type byte is guaranteed not to undergo format conversion.
void†	A type used for operations that return no value, for null pointer constants, and for context handles.
handle_t†	A type used to store binding information in a format that is meaningful to the runtime DCE library but opaque to applications.
error_status_t†	A type used to store DCE status values.
International character types	A set of types constructed from the byte primitive that support international standards for character and string representation.
Constructed Data Types	
Structures	
Unions	This type is somewhat like a C union operation, but embeds a discriminator, which at run time specifies which member of the union is stored.
Enumerations	
Pipes	An open-ended sequence of elements of one type that is used to transfer bulk data.
Arrays	Arrays may be one-dimensional or multidimensional and may be of fixed size, conformant (the array size is determined at run time), or varying (a subset of the array to be transmitted is determined at run time).
Strings	Strings are one-dimensional, null-terminated arrays of characters or bytes.
Pointers	
Context handles	Context handles are not really distinct types, but pointers to void data. They are specified by applying the context_ handle attribute to a parameter. A context handle denotes state information that a server manager will maintain on behalf of a client. Use of a context handle allows this state to persist across several remote procedure calls from the client.

†IDL Keywords

String bindings are easy to generate and manipulate and are suitable for applications in which the user of the client program knows in advance the location of the desired server. The user can supply server location information to the client program interactively or as a command line argument or via a configuration file, and client application code can invoke RPC API routines to compose a string binding and then generate a binding handle that can be passed to the RPC runtime library.

String bindings are well-suited for some RPC applications, but many distributed services require a more flexible and transparent way of establishing bindings. DCE RPC provides an application interface, the RPC name service independent (NSI) API, through which servers and clients can export and import binding information to and from a name service. The use of a name service to store binding information insulates clients from knowledge of where objects and servers reside. The client has only to specify an object and an interface and then use the name service to look up the location of an appropriate server. Thus, the relocation or replication of a server can be made transparent to clients.

As its name suggests, the RPC NSI interface is independent of any particular name service. Thus, applications coded to this interface will potentially be portable across DCE implementations incorporating a variety of name services. In the current HP DCE implementation, DCE CDS underlies the RPC NSI interface, so that the generic RPC name service routines invoke corresponding DCE CDS routines. In principle, another name service such as X/Open Federated Naming (see article on page 28) could supersede CDS in the DCE runtime environment, and existing RPC applications would continue to work.

DCE security, which is described in the article on page 41, is an example of a service that takes advantage of both RPC binding methods. The security client code in the DCE runtime library can bind to a security server either through RPC name service calls or through a string binding generated from a configuration file on the client host. The configuration file solves a bootstrapping problem by making the security service locatable even when CDS is unavailable.

Authenticated RPC. The ability to perform authenticated RPC is crucial to the usefulness of DCE in the real world, where the integrity and privacy of data often must be assured even when the data is transmitted over physically insecure networks. DCE supports several levels of authenticated RPC so that applications will incur only the performance overhead necessitated by the desired degree of protection. These levels include:

- None. No protection is performed.
- Connection. An encryption handshake occurs on the first remote procedure call between the client and the server, exchanging authenticated identities for client and server.
- Call. In addition to connection-level protection, the integrity of the first PDU of each request and response is verified.
- Packet. In addition to call-level protection, replay and misordering detection is applied to each PDU, ensuring that all data received is from the expected sender.

- Packet Integrity. In addition to packet-level protection, the integrity of every PDU is verified. This level can be thought of as protection against tampering.
- Packet Privacy. In addition to packet-integrity-level protection, all remote procedure call parameters are encrypted. This level can be thought of as protection against both eavesdropping and tampering. The privacy protection level is not available in all DCE implementations because of restrictions on the export of encryption technology from the United States.

When data integrity is protected, the sender computes a checksum of the data, encrypts the checksum, and inserts the encrypted checksum in the authentication-verifier portion of the RPC PDU for verification by the receiver. When data privacy is protected, the sender encrypts the actual parameters in the RPC PDU body, and the receiver decrypts them.

The authenticated RPC facility is intended to accommodate more than one authentication and authorization service. A server program registers with the DCE run-time library the authentication protocol it supports. A client specifies an authentication protocol, an authorization protocol, and a protection level in its binding. When the server receives a request, application code in the manager can extract authentication and authorization information from the request. HP DCE currently supports only the shared-secret authentication protocol implemented by DCE security.

Distributed Time Service

The distributed time service, or DTS, is a distributed service that synchronizes the clocks of all hosts in a cell with each other and, optionally, with an external time source. In a typical cell configuration, a few hosts (perhaps three) run a DTS server daemon, and all other hosts run a DTS client daemon called a DTS clerk. One of the DTS server hosts may also run a daemon called a time provider which obtains time from an external source. DTS clerks and servers communicate via RPC and also rely on CDS and security services for naming and authentication.

Clock synchronization is essential for the operation of a DCE cell. The various methods used in several DCE technologies to cache or replicate data, for example, require that clocks agree closely.

In addition to the daemons that synchronize clocks, DTS includes a library of programming interfaces that allow applications to generate and manipulate time values in binary format or in any of several standard textual formats. DTS associates an estimated inaccuracy with every time value, so a time value can also be treated as an interval that is likely to include the correct time. Internally, DTS always keeps time values in the Universal Coordinated Time (UTC) standard governed by the International Time Bureau. The DTS API allows applications to display time values in local time zones.

DTS Clerks. Most hosts in a DCE cell run a DTS clerk. A clerk periodically (at a randomized interval of roughly ten minutes) obtains time values from DTS servers in the cell. The clerk then reconciles these results to compute a single value and inaccuracy that it applies to the local host. This computation takes into account the inaccuracy of each server and an estimate of the time lost to processing and communications. If one DTS server has a faulty clock that disagrees sharply with the others, the clerks will ignore that value, preventing the faulty clock from influencing time throughout the cell. Usually, the time intervals from the servers (time values plus or minus inaccuracies) intersect, and the computed time lies within this intersection (see Fig. 4).

The clerk adjusts time on the local host in such a way that the clock is corrected gradually and continues to advance monotonically. It is especially important to avoid a sudden backward correction because many software systems, including some components of DCE, depend on the monotonicity of the clock. In most computers, a hardware oscillator generates an interrupt at some fixed interval, and this interrupt, called a tick, causes the operating system to advance a software register by some increment. Slight inaccuracies in oscillators cause clocks to drift relative to each other. To adjust time, rather than write the computed correct time directly to the clock register, the DTS clerk changes the increment by which the register advances with each tick. In effect, the software clock rate is increased or decreased to bring the local host into agreement with the servers.

DTS Servers. DTS servers can be configured in two ways:

- If a DTS time provider is running on one of the server hosts, the DTS servers on all other hosts synchronize with the DTS server on that host (roughly every two minutes). Thus, time obtained by the time provider from an external source is propagated to the DTS servers in the cell.
- If there is no DTS time provider in the cell, the DTS servers synchronize with each other (roughly every two minutes). This process is similar to the one used by clerks, except that each DTS server also uses its own time as one of the input values.

External time sources can include telephone and radio services, such as those operated in the United States by the National Institute of Standards and Technology and various satellite services. DTS can also use an Internet network time protocol (NTP) server as an external time source. Though DTS and NTP cannot both be allowed to control the clock on any one client host, the DTS NTP time provider can be used to synchronize a set of DTS-controlled hosts with a set of NTP-controlled hosts.

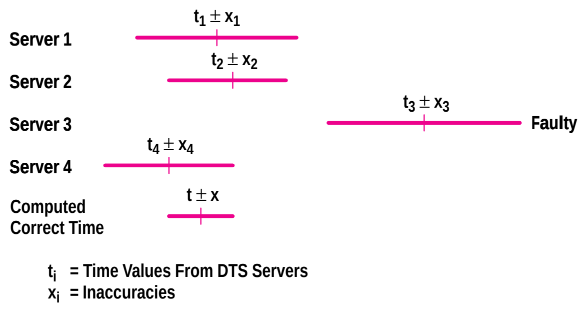

Fig. 4. DTS computing the correct time from several reported times.

DTS servers and time providers attempt to compensate for processing and communications delays when they obtain time values, just as the DTS clerks do.

Distributed File Service

As the UNIX operating system has spread from standalone computers to networked workstations, the need to combine file systems in heterogeneous collections of computers has grown. A few solutions have evolved to meet this need, including the Network File System (NFS) from Sun Microsystems and the Andrew File System (AFS) from Transarc Corporation. The Distributed File Service (DFS) is a successor to AFS that is integrated into DCE.

DFS adds a global filespace to the DCE namespace (see the article on page 23 for a description of DCE naming). Filesets, the logical units of file system manipulation in DFS, are mounted within the DFS filespace for access by DFS clients. DFS cleanly separates the logical and physical aspects of file service, so that a user can always access a file in the DFS filespace by the same name, regardless of where the file or the user physically resides. All DFS file system operations comply with the POSIX 1003.1 specifications for file access semantics. A token-based file access protocol ensures that readers and writers always see the latest changes to a file.

DFS is a distributed service whose major components are a cache manager that runs on DFS client hosts, a fileset server and file exporter that run on DFS server hosts, and a fileset location server that can run on any DCE host. Communication among these components is via RPC; some DFS processes run in the operating system kernel and make use of a special in-kernel implementation of the datagram-based RPC protocol. Fig. 5 illustrates the relationships between these processes and the logical roles that a host can assume. In an actual DFS deployment, one host may play one, two, or all three of these roles.

Other DFS software in the HP DCE product includes a DFS-to-NFS gateway which exports the DFS filespace to NFS, providing secure access to DFS files from hosts outside a DCE cell, an update service that keeps files in synchronization between hosts, a basic overseer server that monitors

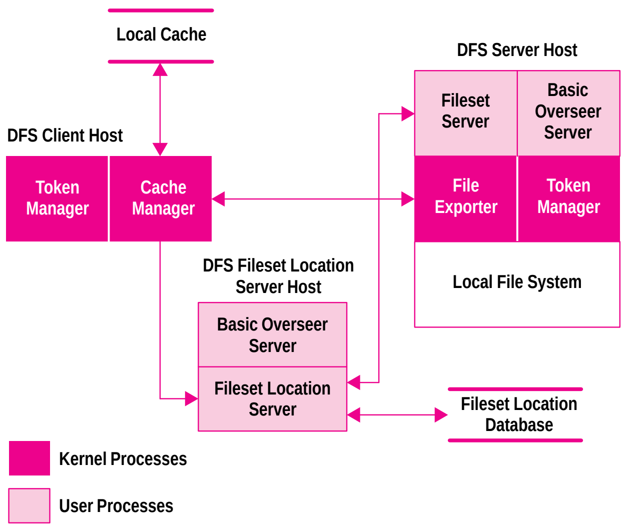

Fig. 5. Relationships between DFS processes.

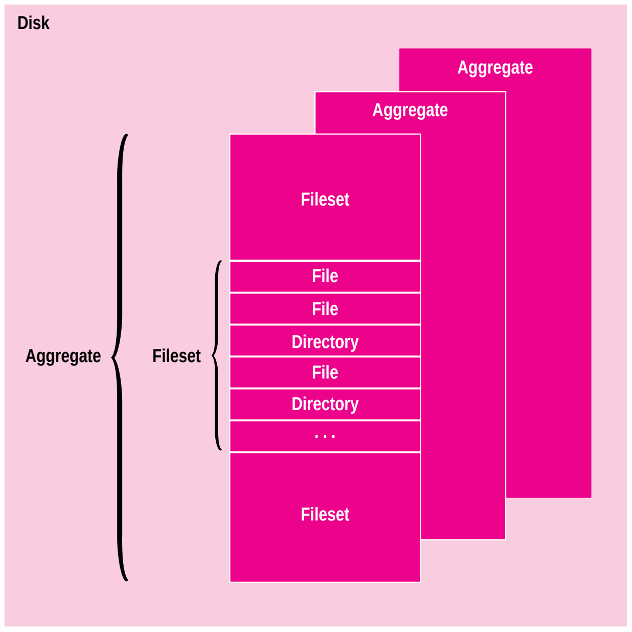

Fig. 6. The relationship between DFS aggregates and filesets.

DFS daemons on each DFS host and supports various remote administrative operations, and other administrative utilities.

Server support for some DFS features is dependent on the level of functionality provided by the local file system software on the server host. For the purposes of this article, a local file system can be classified as either a traditional UNIX file system or an extended file system that offers more advanced functionality. DFS server software can support the full range of DFS features only if the underlying file system provides extended file system functionality. HP-UX file systems currently provide only UNIX file system functionality, so HP-UX DFS server hosts do not support the DFS features that depend on extended file system functionality. If DFS is deployed across a heterogeneous set of platforms, DFS server machines from other vendors may have file systems that do allow full DFS support. When accessing files from such a machine, an HP-UX DFS client host can take advantage of the entire DFS feature set.

Aggregates and Filesets. The DFS filespace is a hierarchy of directories and files that forms a single logical subtree of the DCE namespace. The root of the DFS filespace in a cell is the directory whose global name is /.../<cell-name>/fs. This directory can also be accessed from within the cell by the local name /.:/fs or by the special prefix /:. The directories and files in the DFS filespace can reside physically on many different DFS server hosts in the DCE cell. Two types of DFS resources reside on DFS server hosts: aggregates and filesets (see Fig. 6).

An aggregate is the DFS reference to the physical entity from which one or more filesets are created. From the perspective of the local operating system, this entity is a logical disk managed by local file system software. For example, an aggregate could refer to a logical volume or to a physical partition on a disk.

A fileset is a hierarchy of directories and files that are stored in an aggregate. An extended file system aggregate can store several extended file system filesets, whereas a UNIX file system aggregate can store only one UNIX file system fileset. Each fileset has a name (assigned by an administrator) and a number (generated automatically) that are unique in the DCE cell. A DFS client uses the fileset name to locate the fileset, and thus the files it contains, by looking up the name in the fileset location database.

Many DFS features involve manipulations of filesets. The operations an administrator can perform on a fileset include:

- Mounting it in the filespace so that DFS clients can see its files
- Backing it up
- Setting its quota so that when several filesets reside in one aggregate, the disk space in the aggregate is not disproportionately consumed by one fileset
- Moving it to another aggregate to balance the load among aggregates and DFS server hosts
- Replicating it for performance and reliability.

The last three of these operations are supported only by extended file system aggregates.

Mounting a fileset in the DFS filespace makes the tree of directories and files in the fileset visible to DFS clients. The fileset is mounted at an entry in the filespace, called the mount point, which then names the root directory of the fileset. For example, a fileset containing the home directory for the user Joe might be named users.joe. An administrator might decide to mount the home directories for all users under one directory in the DFS filespace, such as /.../<cell-name>/fs/users. The administrator would issue a command to mount users.joe at, say, /.../<cell-name>/fs/users/joe. Joe could then use this pathname to access his home directory from anywhere. DFS mount points are permanently recorded in the file system as special symbolic links and, unlike UNIX file system mount points, need not be recreated each time a host boots.

A DFS fileset can also be locally mounted, by the UNIX mount command in the directory hierarchy of the local host. For example, the users.joe fileset could be mounted at /users/joe. A file in a DFS fileset thus can be accessed by several names: a local pathname specific to the local host (like /users/joe/mail.txt), a pathname relative to the local cell (like /:/users/joe/mail.txt), and a global pathname (like /.../<cell-name>/ fs/users/joe/mail.txt). DFS guarantees that operations on the file adhere to POSIX semantics, regardless of which way the file is accessed.

DFS Client Components. Each host that accesses the DFS filespace runs a set of DFS client processes that execute in the kernel, which are collectively called the cache manager. The cache manager interacts with the client host kernel, which makes file requests, and with file exporters, which service file requests. It also maintains a local cache of files that have been accessed. The cache can reside either on disk or in memory.

When a file in the DFS filespace is referenced, the virtual file system layer of the kernel invokes the DFS cache manager to handle the reference. The cache manager checks to see whether the local cache can satisfy the requested mode of access to the requested file. If not, it consults the fileset location server to locate the file exporter that manages the requested file and then forwards the request to the file exporter. All data returned by file exporters is cached to reduce load on the servers and on the network.

The interactions of the cache manager with fileset location servers, file exporters, and the local cache are entirely hidden from the operating system on the client host. To the user, accessing a DFS file is no different from accessing a file in a local file system.

DFS Server Components. Each DFS server host runs a set of DFS processes that provide access to its filesets and files.

The fileset server process responds to fileset management requests from administrative clients for filesets residing on the DFS server host. The RPC interface exported by the fileset server includes operations to create and delete filesets, dump and restore them, and get and set status information. Fileset servers cooperate with each other, with fileset location servers, and with file exporters to implement operations such as fileset movement and fileset replication.

The file exporter process responds to file access requests from clients for files residing on the DFS server host. The file exporter is responsible for reading and writing data to the file and for managing attributes of the file such as its modification time.

DFS Fileset Location Server. DFS keeps information about the current state of all filesets in the fileset location database. This replicated database tracks the aggregate and the DFS server host at which each fileset resides. A set of daemons called fileset location servers maintains the fileset location database. Fileset location servers can run on any hosts in a cell but are typically configured to run on a subset of the DFS server hosts.

If a DFS client encounters a DFS mount point while resolving a pathname, it contacts a fileset location server to obtain the current location of the fileset mounted at that mount point. Given the fileset's host and aggregate, the DFS client can then issue a file access request to the correct file exporter. Because clients look up fileset locations dynamically, a fileset can be moved or replicated without users and applications being aware of the change. DFS fileset servers automatically update the fileset location database whenever necessary.

Underlying the fileset location database is a data replication subsystem that implements quorum and voting algorithms to maintain the consistency of fileset location data among all fileset location servers, even in the event of hardware or network failure. A DFS client can thus get current, correct data from any fileset location server.

DFS Token Management

One of the major benefits offered by DFS is its provision of single-site file system semantics. With respect to the file system, programs running on different machines behave in general as though they are all running on the same machine. All clients see a consistent view of the file system. If a process modifies a file in any way, that change is immediately reflected in any operations performed on that file by other processes. To ensure this behavior, each DFS server host must know how clients are using its files. The DFS client and server processes exchange this knowledge and synchronize their actions by exchanging tokens. A token is a guarantee from a server to a client, granting that client permission to use a file from the server in a particular way. Tokens are handled by a DFS subsystem called the token manager, which interacts closely with the cache manager on the client side and the file exporter on the server side.

The following information is encapsulated in a token:

- Token Type. A bit mask that encodes one or more of the values listed in Table II. The token type describes the rights granted to a client by the token.
- File ID. A unique low-level name for a file. It consists of a DCE cell identifier, a DFS fileset identifier, a file identifier, and a version number.
- Byte Range. For data and lock token types, the byte range indicates the portion of the file to which the token applies.

A DFS client cannot perform any operation on a file unless it possesses all the tokens required for that operation. For example, the stat() system call requires a read status token, the read() system call requires both read status and read data tokens , and the open() system call requires an open token. In some cases, the required token is already being held and the operation can proceed immediately. However, in other cases the client machine must contact the token manager on the server host to obtain the necessary tokens.

When the token manager on a DFS server host receives a request for a token from a client, it first decides whether the requested token can be legally granted, based on a set of token compatibility rules. For example, several clients can have read-data tokens for a file, but if one client has a write-data token for a portion of a file, then no other clients can have a read-data or write-data token that overlaps that portion. If the requested token does not conflict with any outstanding tokens, it is granted immediately. Otherwise, the token manager first revokes any conflicting tokens from other clients before granting the requested token.

The rules by which tokens are expired, returned, or revoked are also important for correct semantics and optimal performance. A token has a finite lifetime, which a client can request to extend if necessary. By default, tokens expire after two hours, which is short enough that a token usually will time out before the server has to revoke it, but long enough that the client usually will not need to refresh it. Data or status tokens generally remain with a client until they either time out or are revoked. Before returning a write token, of course, a client must first send back to the server any modifications that it made to the file while it possessed the token.

The file-version information in a token helps clients use cached data efficiently. When a client is granted a token by a server for a file that remains in its cache from a previous access, the client uses the file-version information to determine whether the cached data needs to be obtained again.

Table II
Token Types Used by the Token Manager

Token Type	Rights Granted to a Client
Read Status	Entitles a client to read the attributes of a file and cache them locally
Write Status	Entitles a client to modify the attributes of a file
Read Data	Entitles a client to read some portion of a file designated by an associated byte range and to cache it locally
Write Data	Entitles a client to modify some portion of a file designated by an associated byte range
Read and Write Lock	Indicates that the client has an advisory lock on some portions of a file designated by an associated byte range
Open	Indicates that a process on that client has a file open
Delete	Technically a type of open token which is used during the deletion of files
Whole Volume Token	A special token that applies to the fileset as a whole and is used to coordinate the interaction between ordinary operations on single files and operations on entire filesets, such as the movement of a fileset from one server to another.

Conclusion

The Distributed Computing Environment (DCE) integrates technologies for threads, remote procedure calls, security, naming, time synchronization, and remote file access. DCE eases the development and execution of secure client/server applications and ensures the portability and interoperability of these applications across many kinds of computers and networks.

Acknowledgments
The description of DFS in this article is derived largely from a white paper by John Brezak, Daryl Kinney, Rick Kissel, Charleen Maunsell, Steve Moriarty, and Al Sanders. Parts of the section on threads are based on training materials prepared by Will Hopkins. Rick Kissel, Larry Pare, Al Sanders, Joe Sanfilippo, and Seiichi Tatsukawa provided helpful reviews of this article.

References
1. R. Lalwani, "POSIX Interface for MPE/iX," *Hewlett-Packard Journal*, Vol. 44, no. 3, June 1993.