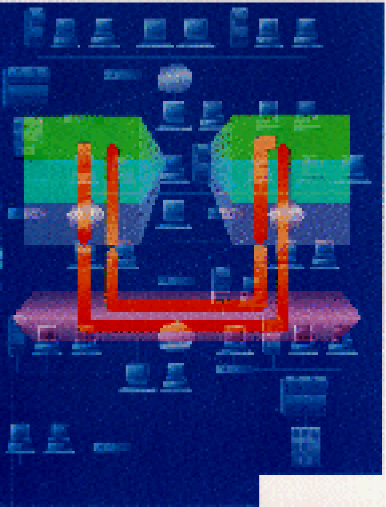
HEWLETT-PACKARD

JOURNAL

[illegible]



[illegible]

table of contents

December 1995,
Volume 46, Issue 6

Articles

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

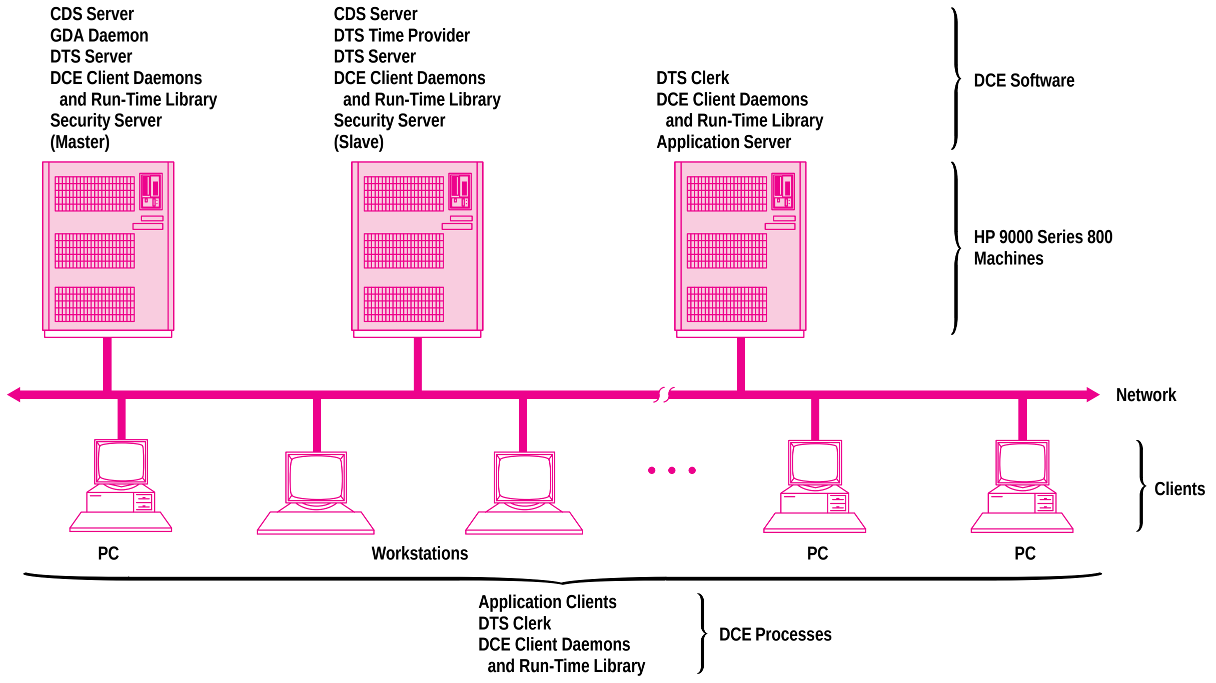


Fig. 1. A single DCE cell containing security, CDS, time, and application servers and application clients running on PCs and workstations.

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.

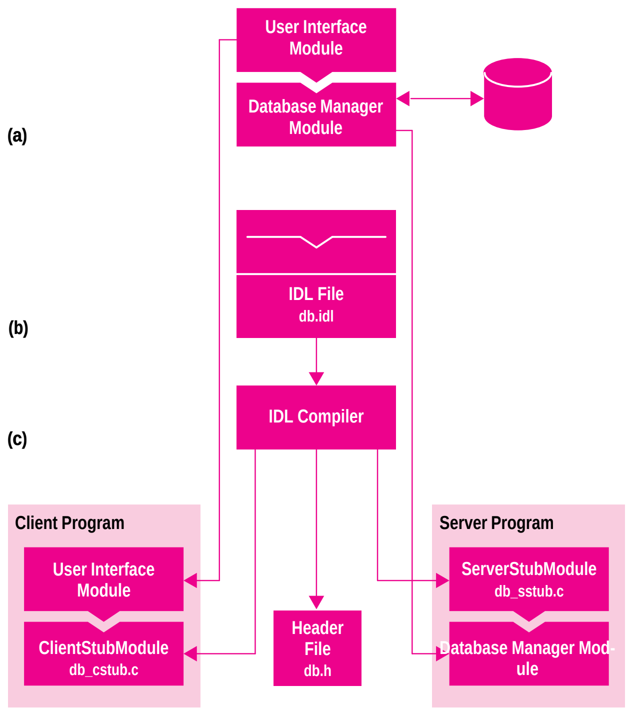


Fig. 2. The process of creating an RPC application. (a) Original application showing the part that will run on the client and the part that will run on a server. (b) Creating an IDL file. (c) Compiling the IDL file to create a header file and client/server stubs.

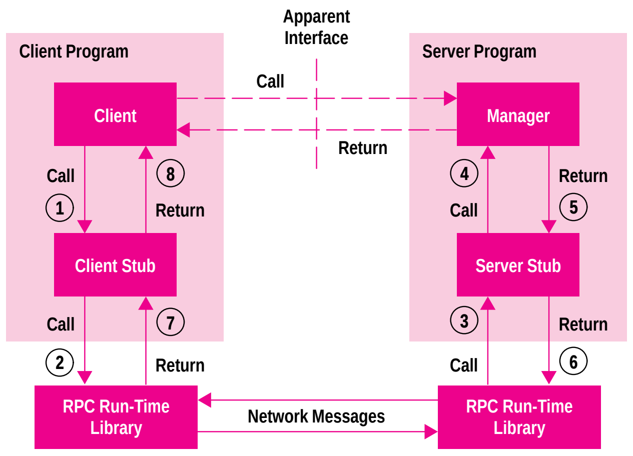


Fig. 3. Flow of events when a client program calls db_lookup on the server.

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 run-time 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.

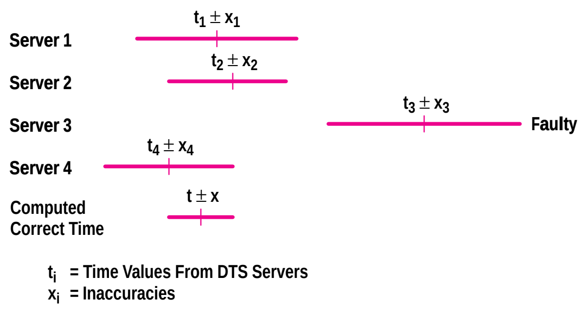


Fig. 4. DTS computing the correct time from several reported times.

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

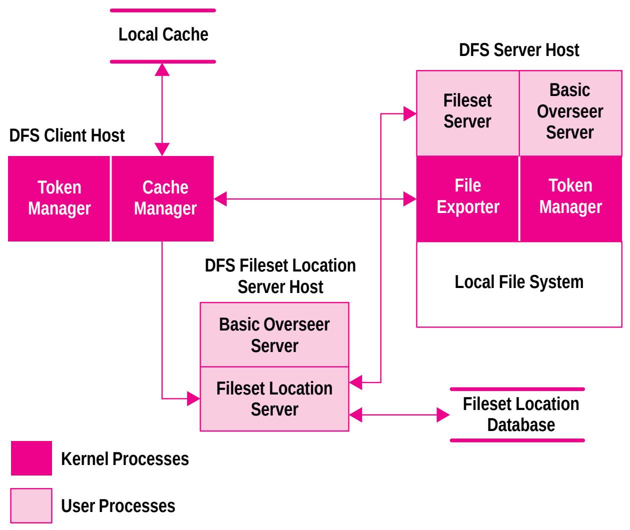


Fig. 5. Relationships between DFS processes.

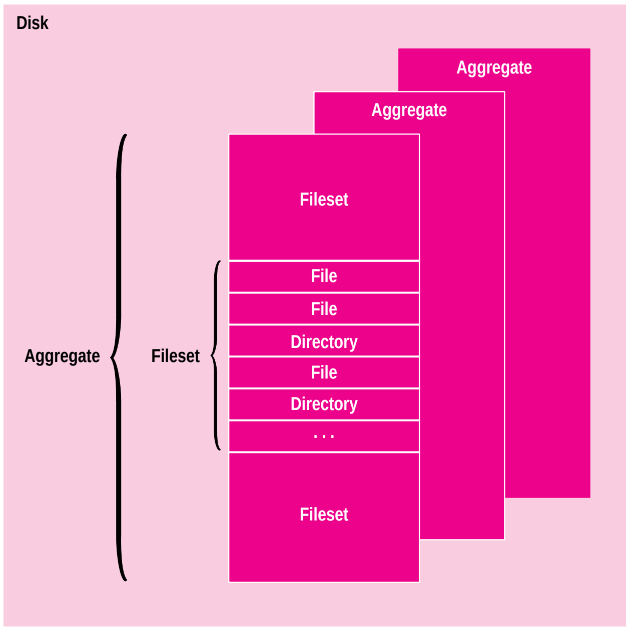


Fig. 6. The relationship between DFS aggregates and filesets.

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.

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.

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.

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.

Adopting DCE Technology for Developing Client/Server Applications

HP's information technology community has adopted DCE as the infrastructure for developing client/server information technology applications. The team developing the DCE service has discovered that putting an infrastructure like DCE in place in a legacy environment is more than just technology and architecture.

by Paul Lloyd and Samuel D. Horowitz

Many companies are navigating the path to open systems. Vendors, including Hewlett-Packard, claim that companies can receive significant benefits by adopting open system client/server approaches for implementing information technology solutions. While the benefits may be attractive, the array of architecture and technology choices is bewildering.

Hewlett-Packard's information technology group has adopted the Open Software Foundation's Distributed Computing Environment (OSF DCE) as a recommended technology for the implementation of client/server applications within HP. The adoption of a technology, or even an architecture, is not sufficient to ensure that the benefits of the client/server model are realized. In fact, once the architecture and technology are chosen, the real journey is just beginning. This paper discusses the issues that led HP to shift toward open systems for information technology client/server applications, the rationale for choosing DCE as a key technology, and the elements of a new infrastructure built to provide the necessary services required to realize the benefits of open systems.

HP's Legacy Environment

Until very recently, HP's legacy environment included multiple mainframes and over 1000 HP 3000 computers operating in more than 75 data centers located around the world.

Business transactions were processed at places called sites, which were major HP installations including manufacturing, sales, and administrative centers. Each site had a local HP 3000 computer system. Most applications were written in COBOL and made extensive use of HP's TurboImage database management system and VPlus/3000 routines for terminal screen management. These tools were used because they made the most effective use of the HP 3000 computer system. At periodic intervals, batch jobs on the HP 3000 systems would create transaction files for transmission to the mainframes. Other batch jobs processed files received from the mainframe. A proprietary store-and-forward system provided the link between the interactive HP 3000s and the batch-oriented mainframes. Fig. 1 illustrates this legacy architecture.

This architecture gave each site access to its own data, but only its own data. Once a transaction was generated and sent to the mainframe, interaction with other production systems meant that response was indeterminate. For example, system users would have to check repeatedly to determine if a purchase order that had been entered was accepted by the factory and scheduled for production. This acknowledgment could take from hours to days depending on the complexity of the order and the number of HP divisions supplying content. In addition, processing and data communication delays anywhere in the company could impact the response time for the transaction, but the user had no way of finding the bottleneck. Further problems

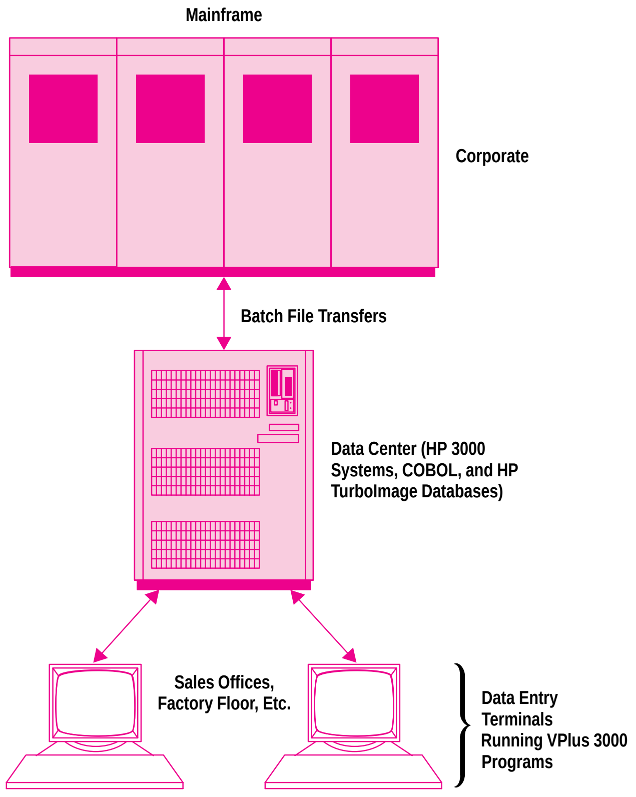

Mainframe
Corporate
Batch File Transfers
Data Center (HP 3000 Systems, COBOL, and HP TurboImage Databases)
Sales Offices, Factory Floor, Etc.
Data Entry Terminals Running VPlus 3000 Programs

Fig. 1 HP's legacy environment for information technology.

were found during massive data center consolidations that have taken place over the past few years.

The architecture for HP legacy applications yielded a large number of applications, each of which required large maintenance and support staffs. Many applications were customized to address the peculiarities of various sites. This contributed to the support problem. As processing power and network bandwidth grew, the customized versions of standard applications made consolidation difficult. In some cases, support costs actually grew.

New applications were both expensive and took a long time to develop, integrate, and deploy. They made little use of previously written code, nor did they share data or other resources effectively. This resulted in a great deal of replicated data within the company. As the environment continued to evolve and new applications came online to address business needs, users found themselves having to manage a multitude of passwords for a large number of systems. From a security standpoint, each password was transmitted multiple times daily over the network, and host-based login services provided the foundation for all data security.

Movement toward Change

Several years ago, HP realized the benefits that could be achieved through open systems client/server architectures. The single biggest driver for the change was a desire to reduce business implementation time, which is the time from when a business need is identified to the time when a production system is in place to address the need. Other drivers included the need for greater cost-effectiveness of the information technology (IT) organization, and the need to reduce operational and administrative costs. An information technology steering group determined that the widespread use of a client/server architecture would enable a reduction in business implementation time and provide increased organizational effectiveness and reduced costs.

A group of IT leaders representing multiple HP organizations formed a task force to develop a client/server architecture for use within HP. Some of the factors the task force considered when choosing the best client/server technology to adopt for our environment included:

- Training optimization and the experience of the current staff
- Concurrent processing in a distributed environment
- Enhancing security for confidential and critical data
- Moving application servers with minimal impact on clients
- Providing interoperability with existing client/server applications and tools
- Enhancing the portability of applications across architectures
- Operating across the HP internet on an enterprise-wide basis.

The evaluation of these factors by the task force resulted in a recommendation that the Open Software Foundation's Distributed Computing Environment be adopted as a standard technology for the implementation of client/server applications within HP.

DCE and the Evaluation Factors

DCE excels in the area of optimizing the training and experience of the current staff. One problem faced by early adopters of client/server computing was that no one could agree on the definition of what was a client and what was a server. This led to a plethora of home-grown and purchased solutions that did little to leverage the nature of the HP computing environment.

The definitions of client and server within DCE are consistent and tangible. A client refers to a program that calls a remote procedure. A server refers to a program that executes the procedure. There is no confusion with hardware or clients and servers on the same system or even a single program being both a client and server. The definitions are entirely consistent. In addition, these definitions fit perfectly within the context of HP's client/server application model.

DCE uses the remote procedure call (RPC) mechanism for client/server communication. This too is beneficial for programmers because it is an extension of a concept that every programmer knows and understands: how to write and execute procedures (or subroutines). In DCE, RPCs behave the same as local procedures. They are still distinct, modular collections of functionality with well-defined parameters that behave in a "black-box" fashion; send them the required parameters, and they reply in a predefined and predictable manner. Further, RPC is unobtrusive in that it hides the complexity of the distributed environment.

With RPCs, application programmers do not need to learn the intricacies of data networking or the particulars of a variety of application programming interfaces (APIs) to implement distributed applications effectively. Unlike other technologies, RPCs ensure that the operational considerations of network programming are both hidden and automatic. Lastly, the DCE APIs required to establish the client/server environment can be easily abstracted to hide even more from the application developer with the further benefit of contributing to consistency in the environment.

Using these concepts, and the tools described later, several of our application teams have experienced reduced implementation times in spite of the need for training in new technologies.

New issues and opportunities arise with the movement to client/server architectures. One of these opportunities is the ability to gain more effective use of computing resources on the network. Through the implementation of a threads facility, DCE gives application developers the ability to have a client call multiple servers simultaneously. In this way, an individual user executing a client program can invoke the parallel processing power of many servers. On the other end, the threads technology also allows servers to respond to multiple clients by processing each request within its own thread. This entails significantly less overhead than the creation and destruction process employed by many alternative technologies that require a unique server process per client. DCE threads are briefly described in the article on page 6.

DCE also incorporates a time service API to provide a consistent network-wide view of time. This service addresses the issues created when applications require time stamps to be reconciled across geographic boundaries or between systems.

Security is another area that raises both issues and opportunities. HP has traditionally used host password security and the security features inherent in the operating system to protect data and applications. If a user gained access to an application, then the user was presumed to have authority to execute any transaction performed by the application. In recent years, some application teams have supplemented host security with features provided by relational database management systems (RDBMS), but these too are usually limited in their flexibility. For example, a user that may have the ability to change a record when executing an authorized transaction should be prohibited from doing so with a database maintenance utility. Such discrimination is beyond the capability of most relational database management systems and requires added attention to system administration.

DCE extends the concept of security to the application itself. The principles of DCE security are authentication and authorization. DCE provides three services to enable the ultimate authorization of actions within a server. The registry service is a database used to keep information about users, groups, systems, and other principals* that can have an identity within the DCE security framework. The authentication service, based on the widely respected Kerberos technology from the Massachusetts Institute of Technology, is used to allow principals to authenticate themselves. The privilege service supplies the privilege attributes for a principal. These attributes are used by an access control list (ACL) manager within the body of a server to make authorization decisions. Using an ACL manager, server authorization decisions can be as granular as business needs dictate. Back doors, such as maintenance utilities or rogue programs, are not possible because users have no access to the systems on which critical data is stored. This makes the properly authenticated and authorized transaction the only vehicle by which a user can affect the database. Security and ACLs are also described in the articles on pages 41 and 49, respectively.

In addition to authentication and authorization, DCE provides features to protect both the integrity and privacy of data transmitted over a network. These features can be invoked by clients or servers when the sensitivity of business data dictates that such precautions are prudent.

Another challenge of the environment is change. Data centers are consolidated and moved, and hardware within the centers is replaced on a regular basis.

DCE provides a flexible, scalable directory service that can be used to apply human readable names to objects such as servers. Servers record their binding information at startup. Clients then locate servers wherever they may be. Multiple directory types permit great flexibility for the application developer. For example, the corporate telephone directory may have replicated instances of the server at many locations. Should one server fail, a client can automatically bind to another. In the case of an online transaction processing system, the one and only server can be found reliably by a client even if it has been moved temporarily after a disaster. Both of these cases can be accomplished with no changes to the client or the user's system configuration. DCE's global directory services are described in the article on page 23.

* A principal can be either a human user or an active object (machine, file, process, etc.).

Hewlett-Packard was a significant contributor to the technology suite that makes up the OSF DCE definition. One of the most important contributions was the RPC mechanism.

DCE's RPC is a compatible superset of the Network Computing System (NCS) from what was once Apollo Computer. The principles upon which the two solutions are based remain the same. They include platform independence and platform unawareness.

DCE platform independence comes from the fact that it runs on all computing platforms in common use within HP's IT environment: HP 3000 computers, HP 9000 workstations, Intel-based Windows® 3.1, and Windows NT. Platform unawareness comes from the fact that application programmers only need to concern themselves with the platform they are working on. Thus, when a developer codes a client, there is no need for the developer to be concerned with what platform the server will run on. Conversely, the server developer does not need to know what platform the client is using. Thus, an application client running on a desktop PC can send a byte string or pointer to a server running on a PA-RISC platform even though the data representations on the two systems are different. Fig. 2 shows a typical configuration of some of the components in HP's DCE client/server environment.

RPC provides the added benefit of interoperating with clients and servers already implemented using NCS. This provides a transition for applications to the more robust world of DCE.

HP operates one of the world's largest private Internet Protocol (IP) networks. The final criterion used by the task force was that the client/server technology chosen must operate

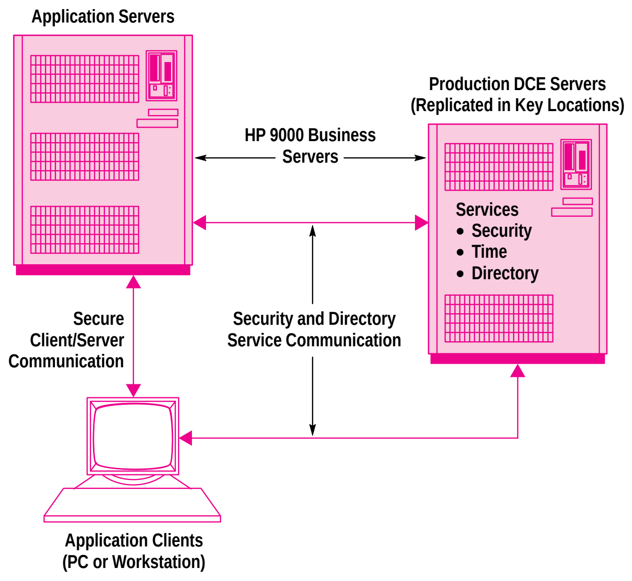

Application Servers
Production DCE Servers (Replicated in Key Locations)
HP 9000 Business Servers
Services
• Security
• Time
• Directory
Secure Client/Server Communication
Security and Directory Service Communication
Application Clients (PC or Workstation)

Fig. 2 The new client/server environment for information technology operations.

on HP's internet in an enterprise-wide fashion. DCE was designed to operate in the IP environment in a scale well above the size required for HP's enterprise.

The task force concluded that the adoption of DCE as a standard technology would enable some significant benefits including:

- Replacement of batch store-and-forward applications with OLTP
- Encapsulation of legacy code and data into servers that can be accessed by GUI clients
- Implementation of client/server applications with minimum training
- Abstraction of much, if not all, of the infrastructure so that application teams can concentrate on the business aspects of applications
- Implementation of enterprise-wide robust security
- Movement of servers between systems without impacting the client or the configuration of the client's host system.

Building HP's DCE Infrastructure

Because of the scope of DCE and the scale of problems that DCE addresses, careful planning is required when deciding how to deploy DCE. We found that the best approach is to start with the customers. As a group responsible for delivering DCE to HP's information technology community, we defined several categories of customers:

- End users. These are the people who interactively use applications.
- Application development teams. These are the people responsible for designing and constructing applications in response to some stated business need.
- Application administrators. These are the people who administer and support business applications in production.

Our group, which is the client/server tools group for HP's corporate network services department, has a long history of providing application data transfer solutions to each of these types of customers. The lessons gained from this experience are simple, even intuitive. End users want technology to be as absolutely invisible as possible, application development teams want to focus on their specific business problems, and application administrators want tools and support. In other words, whenever users must rely on technology to provide a solution, they want to be consumers of the technology and like all consumers, they demand certain things from a technology supplier such as:

- A higher level of abstraction than is usually provided by the technology
- The ability to make necessary, simplifying assumptions
- A consistent level of service in all cases.

Given these requirements, HP has chosen to deploy DCE in the form of an infrastructure. This infrastructure is known as the HP DCE service. In corporate network services parlance a service is an infrastructure with some specific properties. The prime reason for the term service is that the entire effort is focused upon meeting the needs of our customers, the people of Hewlett-Packard. The term service has connotations of careful planning, standardization, published guidelines, and customer satisfaction. It does not have connotations of central control, corporate mandate, or arbitrariness. Finally, a service is a process as much as it is a tangible solution, and it recognizes that as the sophistication of its customers and their business problems grow with time, so too will their expectations.

Associated with every service is a value proposition. The value proposition formally defines the customers, the benefits provided to the customers, and the cost of these benefits.

Our experience designing and constructing a DCE infrastructure can be summarized very succinctly:

- The infrastructure must accommodate diversity.
- The infrastructure must provide consistency.
- The infrastructure must grow experience.

Accommodating Diversity

Identifying diversity is a crucial aspect of customer awareness. The customers of a distributed computing solution come from all parts of the organization with distinct requirements. We identified four categories that the DCE service must accommodate.

The first category is network performance. Customers of the DCE service are, by definition, users of HP's IP network infrastructure. Because of differences in networking technology, customers of the HP internet can realize a difference in both bandwidth and transit delay exceeding two orders of magnitude. Given the request/reply nature of the RPC protocols, this difference will also be reflected in every DCE operation. The lesson of this category is that the infrastructure must not make assumptions regarding the motion of packets.

The second category is application scope. Many business applications are truly enterprise-wide in scope in that there are tens of thousands of clients and massive, dynamic replication of the application servers. Many business applications are only deployed to a single group or department where there are perhaps a few dozen clients and only a single application server is required.

The third category is the different types of application users. Some users use data entry applications in which a small number of transactions are constantly performed to add or modify data. Other users use data query applications to perform a modest number transactions to read data. Other applications provide decision-support services, which typically allow users to perform ad hoc transactions that read data. Finally, some applications serve noninteractive clients that typically invoke large numbers of transactions that read, add, and modify data.

The fourth category is the geographic dispersal of the enterprise. HP does business in several countries all over the world. What this means is that nighttime or weekend service outages cannot be tolerated.

We learned from this kind of analysis that enterprise-wide is a one-dimensional term, and real enterprises are not one-dimensional. We have added the terms enterprise-deep and enterprise-broad. Enterprise-deep addresses the diversity of application users because it acknowledges that a successful infrastructure will accommodate every type of user. Enterprise-broad addresses the diversity of network performance and application scope by acknowledging that all business processing must be accommodated. In addition, today's business processes often cross company boundaries. In these situations, it also necessary to be enterprise-independent.

HP's DCE service accommodates diversity in several key ways. The policies and guidelines for the assignment of registry and namespace (DCE cell directory service) objects support massive replication of application servers. This allows DCE to be used as a foundation for truly enterprise-wide computing. The support model spans all organizations and all time zones, and no customers are ever treated as second class. The subscription model allows all customers to gain access quickly and easily. A subscription-based service provides convenient focal points for satisfying service requests. Finally, policies and guidelines delegate control to the appropriate level. Thus, since DCE is a distributed computing solution, its administration must also be distributed.

Consistency

Providing consistency is a crucial aspect of customer satisfaction. Despite their diversity, all customers are consumers of the same technology. They all demand a complete and comprehensive solution. Furthermore, the biggest return on IT investment comes from building on a consistent foundation that encourages resource sharing and leverages off other infrastructures. As in the case of diversity, the best place to start is with the customers.

End users have specific requirements regarding consistency. Since they must sit in front of and interact with the applications, end users are best served when all applications based on DCE offer a consistent interface with respect to DCE. A consistent interface offers equivalent dialog boxes for performing the standard tasks of DCE login and credential refresh. A consistent interface also offers an equivalent mechanism of dialog boxes or configuration files for server binding and server rebinding. Since DCE is an enabling technology, end users are best served when they can access a variety of DCE applications using their unique, enterprise-wide identity. Gaining credentials should be a side effect of being an employee of the organization, not a side effect of being a user of application X. Finally, there are standard tasks, such as password administration, that all end users must perform and are best served when they all have access to a standard set of tools.

Application development teams have specific requirements regarding consistency. Since application development teams are responsible for incorporating DCE functionality into applications as part of a business solution, they are best served when they can make the necessary simplifying assumptions. Application teams do not want the burden of acquiring and administering the core servers that provide the DCE security service and the DCE cell directory service. Removing this burden is especially important for teams who develop applications that must scale to serve the entire enterprise.

Application development teams also benefit from the ability to use tools that abstract the native DCE APIs. These tools dramatically reduce implementation time, and if they are standard and consistent, training is leveraged across application team boundaries as well as across applications.

Finally, application development teams benefit from the ability to leverage from established best practices and established experts. Despite the common misconception that best practices and experts are an attempt to constrain teams, experience has shown their advantages. Code reuse and resource sharing improve because similarity can be leveraged. Also, business implementation time is less because the need for retraining is reduced, and application quality increases because teams refine, improve, and reuse their skills.

Application administrators have specific requirements regarding consistency. As is the case with application development teams, administrators are best served when they can make the necessary simplifying assumptions. If an end user approaches the administrator to gain access, the administrator should be able to ask the end user's principal name and then perform the appropriate application-specific ACL administration and group management. The nonapplication-specific tasks of requesting an end user principal and machine principal and obtaining properly licensed copies of the DCE software should be left to the end user. The benefit to the application administrator is vastly reduced workload because the administrator only deals with the application. In addition, registry objects such as groups are leveraged across application boundaries.

Application administrators also benefit from the ability to leverage the best practices and standard tools. If DCE applications use DCE objects such as registry groups and namespace entries in a consistent fashion, retraining is minimized and a large cause of administrative errors is reduced.

Hewlett-Packard's DCE service provides consistency in many ways. Cell boundary decisions are weighted in favor of larger cells to promote genuine enterprise-wide computing. Tasks associated with DCE cell administration have been abstracted into high-level tools based upon the service's subscription model. These tools automate and hide specific, low-level DCE tasks. For example, the task that corresponds to an application subscription creates principals, groups, and accounts for the application's servers, creates namespace entries for the application, and modifies all associated access control lists. The benefit of this abstraction is the consistency that it ensures because the actual registry and namespace objects are generated and administered in a standard, documented manner.

Growing Experience

Growing experience, which means making both application developers and application users successful, is a crucial aspect of realizing the business benefits of DCE. Clearly, an infrastructure that is not used is useless. Growing experience is a two-step process that never ends. The first step is to identify barriers, and the second step is to remove these barriers by any means necessary. Such means include, but are not limited to, the development and deployment of custom tools, the abstraction of DCE tasks to better suit existing business practices, and exploitation of the fact that DCE is already one of its own best customers. The need for custom tools is by no means a negative reflection on DCE, but simply an acceptance of the fact that no single solution can do everything for everybody. Abstraction is simply a way to make DCE fit the business rather than forcing the business to fit DCE. Taking advantage of DCE means understanding that everything in DCE is basically a DCE object accessed by a client through an interface and protected by an ACL.

Application developers face a variety of barriers. The most traumatic barrier stems from the large number of new technologies directed towards development teams. Keep in mind that in most organizations, new technology really means new to the organization. In Hewlett-Packard, most IT application teams are new to writing distributed applications using DCE's client/server split or RPC paradigm. Our distributed applications have traditionally been based on file transfers or message passing. The learning curve for all of the technologies associated with DCE is nontrivial, especially when the development tools associated with the technologies are still evolving. The consequence to application developers is that creating the first DCE application with out-of-the-box DCE, even an evaluation application, is usually a difficult task. The risk is that IT application teams will not consider using DCE.

Application developers also face barriers when testing or deploying applications. The richness of DCE offers the developers an often bewildering variety of choices such as different ways to take advantage of the namespace or different methods of allocating registry objects to take advantage of DCE security. Without guidelines, established practices, and assistance some teams will simply try anything and then fail. Reports of these failures usually travel faster and wider than reports of teams that succeed.

HP's DCE service removes these barriers in three key ways. First, the service provides a DCE development library that abstracts the native DCE APIs into two very high-level API routines that include one call for the client and one call for the server. Second, the service offers a custom version of the OSF DCE programmer's class, which focuses on HP's IT environment. Third, the service offers consultants who can help other entities start DCE projects. A typical consulting venture involves the creation of Interface Definition Language (IDL)† files, a skeleton server that takes full advantage of security and the namespace, and a skeleton client that can bind to the server. After this is all done the application team only has to add the application code between the curly braces. The best part of DCE is that it allows one to distribute an application without worrying about how to do it.

Application administrators face barriers because for DCE applications, there will be DCE related tasks that they must perform either directly or indirectly in a production environment. Although production-quality DCE applications do not require much attention, there are still issues that can arise. For example, there is the occasional administration of endpoints and namespace entries in server failure cases, the occasional administration of server keytab files, and most of all, the administration of the application's ACLs used to control authorization. The out-of-the-box DCE tools are cumbersome and error-prone, and worst of all, they are fairly low-level and require a fairly detailed knowledge of DCE concepts. The risk is that DCE applications can acquire an undeserved reputation as being costly and difficult to support in production.

† IDL is a language similar to C that allows developers to specify the interfaces that tie client and server applications together.

HP's DCE service removes these barriers by providing custom OSF/Motif tools to ease these DCE related tasks. Also, the published guidelines and best practices that bring consistency to DCE applications can help to grow experience by reducing the need to retrain.

System administrators face barriers because of the complexity of one of the most common tasks in a growing, maturing DCE cell. Since DCE regards each physical machine as a principal with its own authenticated identity, configuration must be done on each machine when adding it to the cell. The out-of-the-box tools have two significant problems. First, they require coordination by the system administrator for root access. Second, if configuration is done across the network, both the root password and the DCE cell_admin password are exposed. These are unacceptable security holes especially for a system that is intended to serve as a foundation for secure distributed applications. In addition, many machines already run NCS applications, and these applications must not be impacted by the migration to DCE. As a result, installation and configuration are tedious and potentially insecure. The risk is that deploying DCE throughout the enterprise will be viewed as slow and expensive.

Hewlett-Packard's DCE service removes these barriers in three key ways. First, we have developed a scheme that allows a machine to be remotely and securely added to a cell. In particular, this scheme does not expose the operating system or DCE passwords across the network. It also doesn't require any effort on the part of the system administrator other than to install an HP-UX* fileset. Second, we are integrating the DCE client software into a common operating environment for machines that run the HP-UX operating system. Third, we provide a PC-DCE†† license server to ease the distribution of PC-DCE.

End users face a variety of potential barriers. Although it is the job of the application developer to shield DCE from end users, they will still be aware of their DCE principal. Thus, end users should have minimal training on obtaining and refreshing their network credentials as well as managing their principal. Out-of-the-box DCE does not include a standalone password management tool, and users must actually run rgy_edit and modify their account. Also, integrated login solutions in which the operating system and DCE logins are combined are still evolving (see the article on page 34). The risk is that deploying DCE applications to large numbers of end users can be slow, tedious, and expensive, and the end users who are exposed to too much DCE because of poorly constructed applications may assume that all DCE applications are difficult to use.

HP's DCE service removes these barriers in two key ways. First, we provide tools on both the HP-UX operating system and the PC to ease password administration. Second, the service's subscription model provides a simple focal point for requesting and obtaining a DCE principal.

†† PC-DCE is an implementation of DCE that runs in an MS Windows environment.

The final barrier to growing experience comes from two groups of people. The first group believes that client/server is really just remote SQL, and the second group believes that client/server just means the motion of bits over the network. Remote SQL is certainly a fine solution for some business problems. However, it is important to remember that vendor lock-in, client-side awareness of database schema, network performance on the WAN, and the usual lack of network security could be problems in dealing with remote SQL. Although DCE does move bits over the network, and other approaches such as message passing using sockets may be an adequate solution for many business problems, the issues of WAN performance, architecture differences, code sharing difficulties, code maintenance difficulties, and the usual lack of network security could be problems in other approaches. When making technical decisions, being dogmatic is usually the first step towards being unsuccessful. The goal is not to dictate or even to impress, but to educate and promote a community in which decisions are made objectively.

Hewlett-Packard's DCE service addresses these barriers in two ways. First, we offer classes on all aspects of DCE and its use. Second, service subscribers can access all published information using Worldwide Web browsers.

Conclusion

Perhaps the best way to get a clear perspective of HP's DCE service is via analogy with other well-known infrastructures. Consider customers of a WAN. Everyone wants access and a consistent service model such as enterprise-wide IP connectivity. Consider the technology that is used to build a WAN. Now consider a successful WAN infrastructure. It is much more than the technology (i.e., routers, bridges, etc.) used to build it, it is a also a distributed creature that requires distributed administration and coordinated planning and guidance. Furthermore, there is no distinction between a test network and a production network. The network is simply an infrastructure that supports all phases of the software lifecycle.

Another valuable analogy is the interstate highway system in the United States. In the 1950s automotive technology boomed and the resulting cost structures allowed many people to own a car. This produced a fundamental change in American society because of the freedom, power, and movement of goods and services the automobile permitted. Perhaps the biggest contributing factor was the interstate highway system. The interstate highway system really wasn't about automotive technology. It was about use and access. Distributed applications are the same. The focus shouldn't be on distributed computing technology but on use and access.

DCE is a powerful and impressive collection of software technology. It offers attractive solutions to the kinds of business problems that most large organizations must address. Our experience has demonstrated the following:

- It is OK to experiment.
- It is important to allow a few key people to become industry-level experts. These are the people who should be responsible for service management.
- DCE should not be managed by regulatory practices.
- It is imperative to abstract anything if the result better fits the business needs and business practices.
- Activities should not be done in secret or kept secret.
- A service such as DCE is as much a continual process as it is a tangible solution.

HP-UX 9.* and 10.0 for HP 9000 Series 700 and 800 computers are X/Open Company UNIX 93 branded products.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

X/Open is a registered trademark and the X device is a trademark of X/Open Company Limited in the UK and other countries.

Open Software Foundation, OSF, and OSF/Motif are trademarks of the Open Software Foundation in the U.S. and other countries.

Windows and MS Windows are U.S. registered trademarks of Microsoft Corporation.

DCE Directory Services

The DCE directory services provide access for applications and users to a federation of naming systems at the global, enterprise, and application levels.

by Michael M. Kong and David Truong

The directory services of the Open Software Foundation's Distributed Computing Environment (DCE) enable distributed applications to use a variety of naming systems, both within and outside a DCE cell. Naming systems make it possible to give an object a textual name that is easier for humans to use—easier to read, pronounce, type, remember, and intuit—than an identifier such as a DCE universal unique identifier (UUID). Information about the locations of objects can be stored in naming systems so that users can access an object by name, with no a priori knowledge of its location.

DCE provides three directory services:

- The Global Directory Service (GDS) is the DCE implementation of the CCITT (International Telegraph and Telephone Consultative Committee) 1988 X.500 international standard. GDS is a distributed, replicated directory service that manages a global namespace for names anywhere in the world.
- The Cell Directory Service (CDS) is a distributed, replicated directory service that manages names within a DCE cell.
- The Global Directory Agent (GDA) is a daemon that uses global name services to help applications access names in remote cells. GDA interacts with either X.500 services such as GDS or Internet Domain Name System (DNS) services such as the Berkeley Internet Name Domain (BIND) name server, named.

Through these services, DCE applications can access several interconnected namespaces, including X.500, DNS, CDS, the DCE security namespace (see article, page 41), and the DCE Distributed File Service (DFS) filespace (see article, page 6).

The DCE Namespace

The DCE namespace is a federation of namespaces at three levels: global namespaces, an enterprise namespace, and application namespaces. A DCE name can span one, two, or all three of these levels. GDS and BIND name servers provide X.500 and DNS global namespaces in which the names of DCE cells are stored. Within each DCE cell, CDS provides an enterprise namespace, and the names of CDS objects in that namespace are relative to that cell. At the application level, DCE subsystems including DCE security and DFS define their own namespaces.

DCE names are hierarchical names consisting of a series of components delimited by the / character. The first component of a DCE name is one of three prefixes denoting the root of a namespace:

- /... is the global root. A name that begins with /... is called a *global name*.
- /.: is the root of the local cell. This prefix is shorthand for /.../<local-cell-name>. Names that begin with /.: are called *local names*.
- /: is the root of the DFS filespace. This prefix, shorthand for /.:/fs, makes DFS filenames easier to use.

Within the local DCE cell, local and global names for an object are equivalent and interchangeable. However, when a user references a local name, the resolution of the name is relative to whatever cell the user is in. Hence, to access an object in a remote cell, a user must refer to the object by its global name.

A DCE cell has a global name that may be either an X.500 name stored in GDS or a DNS name stored in a BIND database. For example, a cell at HP's Cupertino site could have the GDS global name /.../c=us/o=hp/ou=cupertino. In X.500 syntax, the components of a name are separated by the / character, and each component describes an attribute of the object. The component o=hp, for instance, signifies the organization named HP. The DNS global name /.../cssl.cell.ch.hp.com might name a cell in HP's Chelmsford systems software lab (CSSL). As this example shows, a DNS name is a single component of a DCE name but is itself a compound name. DNS names are made up of names in the hierarchical DNS namespace, separated by periods and ordered right-to-left from the DNS root.

Objects in a cell have names that are composed of the cell name, a CDS name, and possibly a name from an application namespace. Some objects, such as RPC servers, are named directly in the CDS namespace; their names consist of a cell name plus a CDS name. Other objects, such as DFS files or DCE security principals, are managed by a service that implements an application namespace. The name for such an object is formed by concatenating a cell name, a CDS name for the root of the application namespace, and an application name relative to that root. For example:

- If the HP Chelmsford systems software lab cell is running an RPC server from the Acme Database Company, that server might be registered under the name /.../cssl.cell.ch.hp.com/ acme/acme_server (see Fig. 1). Within the CSSL cell, /.:/acme/acme_server would be an equivalent name for the server.
- If the HP CSSL cell includes a principal named Nijinsky, that principal would have the global name /.../cssl.cell.ch.hp.com/sec/principal/nijinksy (see Fig. 2) and the local name /.:/sec/principal/nijinksy.
- If the DFS filespace in the HP Cupertino cell contains a file called /users/sergey/mail/igor, that file would have the global

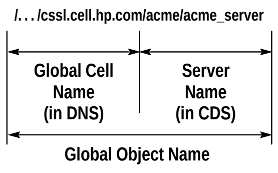

/. . . /cssl.cell.hp.com/acme/acme_server
Global Cell Name (in DNS)
Server Name (in CDS)
Global Object Name

Fig. 1. A DCE global name for a server.

name /.../c=us/o=hp/ou=cupertino/fs/users/sergey/mail/igor (see Fig. 3). Within the Cupertino cell, the names /.:/fs/users/sergey/mail/igor and /:/users/sergey/mail/igor would be equivalent names for the file.

Directory Service Interfaces

DCE offers two sets of programming interfaces to directory services. The RPC Name Service Independent (NSI) API is a generic naming interface provided by DCE RPC. The X/Open® Directory Service (XDS) and X/Open OSI Abstract Data Manipulation (XOM) APIs are interfaces based on CCITT X.500 standards.

Transparently to an application, the DCE directory services interact with each other as necessary to resolve names. Fig. 4 illustrates some of the interrelationships between these services.

When a DCE application passes a name to the RPC NSI API, CDS client software (in the DCE run-time library and in CDS client daemons) uses DCE RPC to contact a CDS server either in the local cell or in a remote cell to look up the name. Names in the local cell are passed directly to a CDS server in the cell. Names in a remote cell are passed to a GDA daemon, which performs a lookup in X.500 or DNS, depending on the syntax of the cell name, to obtain the location and identity of a CDS server in the remote cell. The CDS client software then uses this information to contact the remote CDS server.

When an application passes a name to the XDS/XOM APIs, the XDS code in the DCE run-time library resolves the name according to its syntax. If the name consists purely of components such as c=us and o=hp, the XDS library passes the name to the GDS client code, which contacts the GDS server to look up the name. If any portion of the name is not in GDS syntax, the name is passed to the CDS client code and resolved in the same way as names passed through the RPC NSI API.

GDS Directory Structure

The GDS directory is a collection of information about objects that exist in the world. Information about objects is stored in a database called a *directory information base*. A directory information base contains an entry that completely describes each object and may also contain an alias entry that provides an alternative name for an object entry.

An entry in the directory information base consists of a set of attributes, each of which stores information about the object to which the entry refers. An attribute is made up of an attribute type and one or more attribute values. For example, an entry for a person might include attributes whose attribute types are surname, common name, postal address, and telephone number. Attributes that have more than one value are called multivalued attributes. A person with more than one telephone number would have a multivalued telephone number attribute.

Each entry can belong to one or more object classes. An object class of an entry restricts the permitted attributes for that entry. The mandatory and optional attributes of entries in an object class are determined by object class rules, and

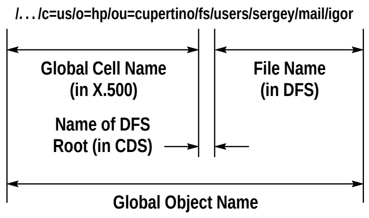

/. . . /c=us/o=hp/ou=cupertino/fs/users/sergey/mail/igor
Global Cell Name (in X.500)
Name of DFS Root (in CDS)
File Name (in DFS)
Global Object Name

Fig. 3. A DCE global name for a file.

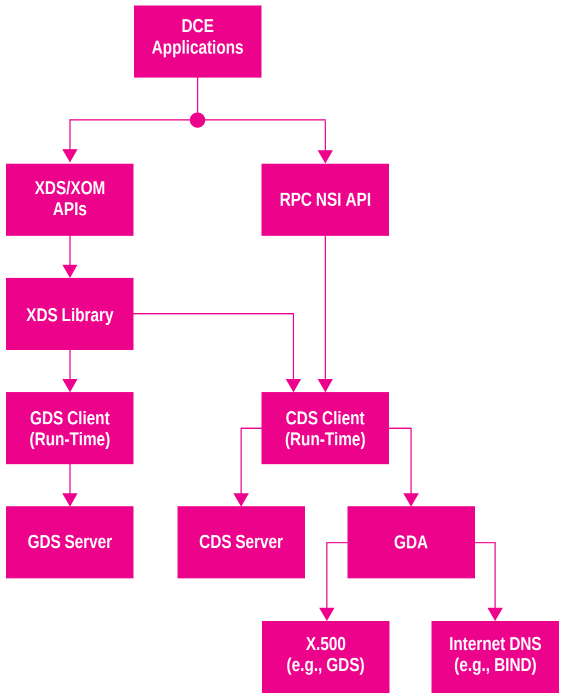


Fig. 4. Interrelationships between DCE directory services. The application program interfaces (APIs) are the DCE remote procedure call name service independent API (RPC NSI API) and the X/Open Directory Service (XDS) and X/Open OSI Abstract Data Manipulation (XOM) APIs. GDS is the DCE Global Directory Service and CDS is the DCE Cell Directory Service. GDA is the DCE Global Directory Agent. X.500 is an international standard implemented by the DCE GDS. DNS is the Internet Domain Name System. The Berkeley Internet Name Domain (BIND) is an implementation of DNS.

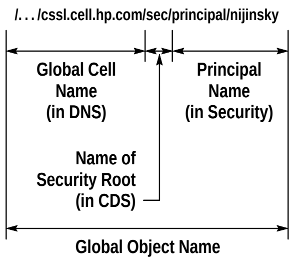

/. . . /cssl.cell.hp.com/sec/principal/nijinsky
Global Cell Name (in DNS)
Name of Security Root (in CDS)
Principal Name (in Security)
Global Object Name

Fig. 2. A DCE global name for a principal.

these rules are part of a schema. For example, an entry representing an organization must contain an attribute called Organization-Name, which has the name of the organization as its value. An entry can contain optional attributes that describe the organization: the state or locality in which the organization resides, the postal address of the organization, and so on. As a general rule, all entries must contain the Object-Class attribute, which contains the list of object classes to which the entry belongs. If an entry belongs to more than one object class, all object classes must be listed in this attribute.

As discussed above, attribute types and object classes have human-readable names that are meaningful and unique, but they are not used in the protocols; an object identifier is used instead. An object identifier is a hierarchical number assigned by a registration authority. The possible values of object identifiers are defined in a tree. The standards committees ISO and CCITT control the top of the tree and define portions of this tree. These standards committees delegate the remaining portions to other organizations so that each object class, attribute type, and attribute syntax has a unique object identifier. For example, the object identifier of the country object class is 2.5.6.2, which can also be written more verbosely as:

joint-iso-ccitt{2}modules{5}object classes{6}country{2}.

X.500 Naming Concepts

Information in the directory information base is organized in a hierarchical structure called a directory information tree. A hierarchical path, called a *distinguished name*, exists from the root of the tree to any entry in the directory information base. Each entry in the directory information base must have a name that uniquely describes that entry. For example, the employee (entry) David has the distinguished name C=US/O=hp/OU=hpind/CN=David, where C denotes the country, O the organization, OU the organization unit, and CN the common name.

The distinguished name is a collection of attribute type and attribute value pairs called *relative distinguished names*. From the example above, C (country) is an attribute type and US (United States) is an attribute value.

Alternative names are supported in the directory information tree through special entries called *alias entries*. An alias entry is a leaf entry in the directory information tree that points to another name. Alias entries do not contain any attributes other than their distinguished attributes because they have no subordinate entries.

GDS Components

As shown in Fig. 5, GDS is made up of four main components:

- Directory User Agent (DUA). This process is the user's representative to the directory service. The user can be a person or an application.
- Directory System Agent (DSA). This process controls and manages access to directory information.
- DUA Cache. This process keeps a cache of information obtained from the directory DSAs. One DUA cache runs on each client machine and is used by all the users on that machine. The DUA cache contains copies of recently accessed object entries and information about DSAs.
- Directory Information Base. This is where GDS stores information.

The DUA and DSA communicate by using the directory access protocol. DSAs use the directory system protocol to communicate with each other.

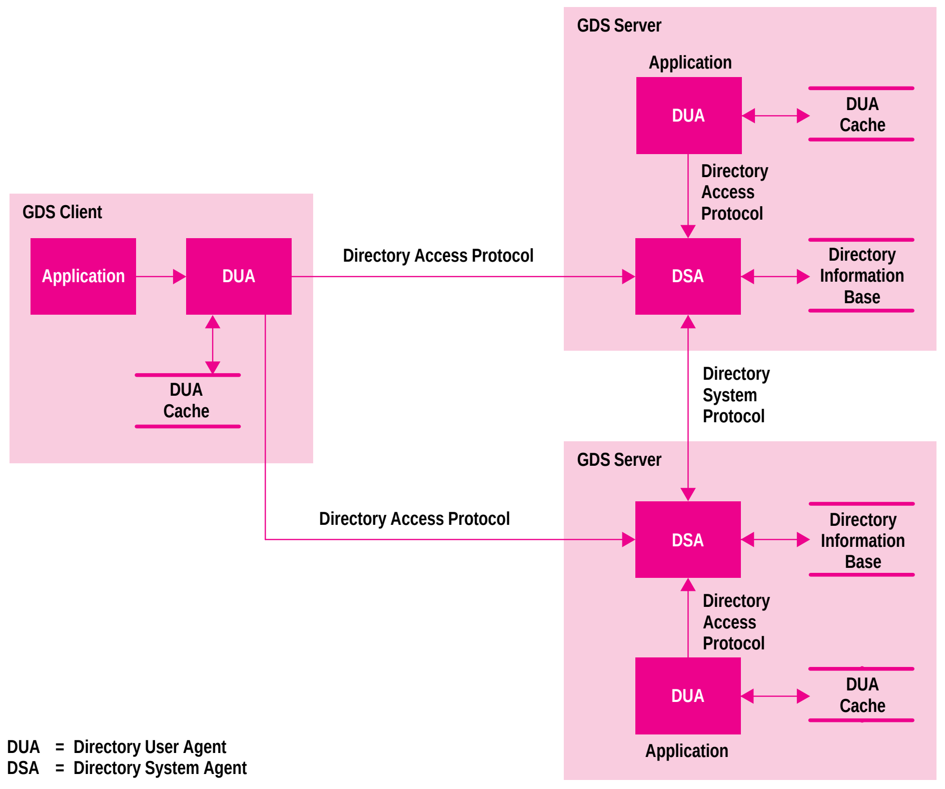

GDS Server
Application
DUA
DUA Cache
Directory Access Protocol
GDS Client
Application
DUA
Directory Access Protocol
DSA
Directory Information Base
DUA Cache
Directory System Protocol
GDS Server
Directory Access Protocol
DSA
Directory Information Base
Directory Access Protocol
DUA
DUA Cache
Application
DUA = Directory User Agent
DSA = Directory System Agent

Fig. 5. Global Directory Service (GDS) components.

Since directory information is distributed over several DSAs, a DUA first directs any queries to a specific DSA. If this DSA does not have the information, there are two standard request methods that the DUA can use. The first method is referral—the DSA addressed returns the query to the DUA together with a reference indicating the other DSAs that have the information. Chaining is the second request method—the addressed DSA passes on the query directly to another DSA via the directory system protocol.

CDS Directory Structure

Every DCE cell must have at least one CDS server. The CDS servers in a cell collectively maintain a cell namespace, organized into a hierarchical directory structure. As mentioned above, the prefix /.: is shorthand for the global name of the cell and hence denotes the root of the cell namespace.

A CDS name is simply a series of directory names followed by an entry name. The directory names are ordered left-to-right from the cell root and are separated by the / character. For example, in the name /.:/acme/acme_server, the directory acme is a child of the cell root and the object acme_server is an entry in acme.

In a cell that contains more than one CDS server, CDS directories can be replicated, with each replica of a directory managed by a different CDS server. Among the replicas in a set, only one, the master replica, is modifiable; all other replicas are read-only. Replication increases the availability and reliability of CDS service and can ease recovery from hardware or network failure.

A CDS directory can contain three types of entries:

- Object entries contain information about objects in the cell. An object can be a host, a user, a server, a device, or any other resource that has a CDS name.
- Soft links provide alternate names for objects, directories, or other soft links.
- Child pointers are pointers to the directories contained by a parent directory. A child pointer is created when a new directory is created and is used by CDS to locate replicas of that directory when resolving the directory's name. Child pointers are created only by CDS itself, not by applications.

Like GDS, CDS stores information about named objects by associating attributes with names. Object entries might have attributes to store the object's UUID, its location, its access control list (ACL), the time it was created, and the time it was last updated. A soft link has an attribute to store the name of the object to which the link points.

Two special classes of CDS object entries warrant particular mention:

- RPC server entries store information about servers, including their location and the objects they manage, in the CDS database. Servers register this binding information, and clients look it up, via the RPC NSI interface.
- Junctions enable a service that implements its own namespace to splice that namespace into the DCE namespace. A junction is somewhat analogous to a mount point in a UNIX® file system; the junction entry stores binding information for a service and becomes the root for the namespace managed by that service. The CDS entry /.:/sec, for example, is the junction for the DCE security service. Applications can use names such as /.:/sec/principal/stravinsky to identify principals in the security registry and to obtain bindings to a security server. Similarly, /.:/fs is the junction for DFS, and /.:/hosts/<host-name>/config is the junction for the configuration services provided on each host by the DCE host daemon, dced.

CDS Components

CDS is a distributed service based on a client-server model. Fig. 6 illustrates the software components that implement this service.

All CDS directory data is stored in databases called *clearinghouses*, which are managed by CDS server daemons. The server daemon responds to requests from CDS clients for lookups in or updates to the database. When an RPC server invokes an RPC NSI API routine to export binding information to the namespace, for example, this routine triggers a CDS update operation. Similarly, when an RPC client imports bindings from the namespace, a CDS lookup operation is executed. Each CDS server keeps an image of its clearinghouse in memory and writes the clearinghouse periodically to disk.

A cell often includes more than one CDS server, each with its own clearinghouse. Running several CDS servers in one cell allows administrators to replicate CDS directories. If a directory is replicated, one clearinghouse stores the master replica of the directory, and other clearinghouses store read-only replicas. Clients can perform lookups from any replica but can perform updates only to the master replica. After a CDS entry is updated in the master replica of its directory, the CDS server that manages the master replica propagates the update to all CDS servers that manage read-only replicas. Replication improves responsiveness to clients by distributing work among several servers and ensures the availability of CDS service if a server machine fails or the network fails.

The architecture of CDS insulates applications from direct communication with CDS servers. To add, delete, or modify CDS data, applications call APIs such as the RPC NSI routines in the DCE run-time library. CDS client code in the library interacts with a daemon on the local host called the

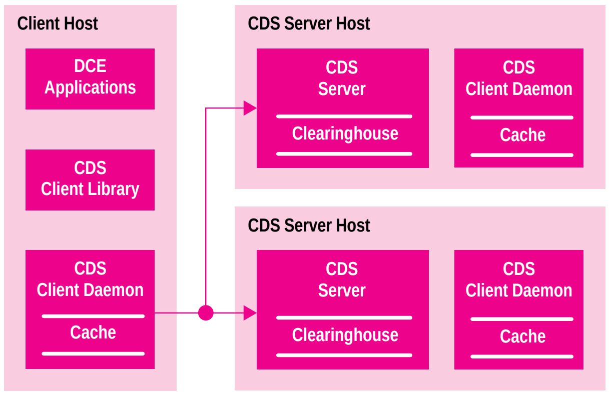


Fig. 6. Cell Directory Service (CDS) components.

CDS advertiser, which uses RPC to communicate as necessary with CDS servers. A CDS advertiser runs on every host in a DCE cell. (In many DCE implementations, several CDS client daemons execute on each host: one CDS advertiser and a number of CDS clerks. In the HP DCE/9000 product, a single CDS advertiser process subsumes all advertiser and clerk tasks.) To increase client performance, reduce server load, and reduce network traffic, each advertiser saves the results of its lookups in a cache. Frequently accessed data can be retrieved locally from the cache rather than via RPC from a server. The advertiser writes the cache to disk periodically, so cached data persists through reboots of the CDS client host.

Conclusion

DCE provides a three-level naming system and two naming APIs.

Names of cells are stored in a global namespace managed by a DNS server such as named or by an X.500 server such as the DCE Global Directory Service (GDS). The Global Directory Agent (GDA) oversees resolution of global cell names.

The cell namespace consists of two levels: the enterprise namespace managed by the DCE Cell Directory Service (CDS) and application namespaces such as those managed by DCE security and the DCE Distributed File Service (DFS). The roots of application namespaces are named by CDS junctions.

DCE offers two naming APIs. The RPC NSI interface is used by servers to register their names and locations in CDS and by clients to look up names and get back binding information. The XDS/XOM API can access names and their associated attributes in GDS and CDS.

The DCE name services have some limitations that X/Open's Federated Naming specification attempts to solve (see article, page 28). The RPC NSI API is a specialized interface that manages only RPC binding handle information; it cannot read or manipulate other attributes associated with a name. Many programmers find the XDS/XOM API cumbersome; this interface is also difficult to layer over other existing naming APIs. The RPC NSI API and the XDS/XOM API do not offer a way to create or delete directories programmatically, so an application that needs to create directories currently must use an internal CDS interface. The CDS and GDS protocols are complicated and not very general. New naming services that are introduced are unlikely to use either of these protocols or the XDS API. Finally, CDS does not support an easy, general way to create and resolve through junctions to application namespaces.

Acknowledgments

Liza Martin and Paul Smythe each reviewed several drafts of this article and made valuable suggestions, for which we are grateful.

References

X/Open Federated Naming

The X/Open Federated Naming (XFN) specification defines uniform naming interfaces for accessing a variety of naming systems. XFN specifies a syntax for composite names, which are names that span multiple naming systems, and provides operations to join existing naming systems together into a relatively seamless naming federation.

by Elizabeth A. Martin

Naming of objects is a fundamental need in a computing system. A naming service maps human-readable names to internal location information that programs use to access the named objects. Current distributed computing environments that take advantage of large computer networks present new problems and requirements for the naming service.

Heterogeneous naming systems are a reality. Unlike the naming service in a single-host system, the naming service in a distributed system is usually not a monolithic component but consists of various naming systems embedded in different pieces of the system. The naming systems in the Open Software Foundation (OSF) Distributed Computing Environment (DCE, see article, page 6) include the X.500 directory service,[1] the DCE Cell Directory Service (CDS),[2] and the DCE Distributed File System (DFS, see page 6). The DCE security service (see article, page 41) and the DCE daemon (see article, page 6) also support namespaces. A typical DCE installation will have applications that have their own naming systems, such as databases, email, desktops, and spreadsheets.

These different naming systems have arisen in part because they meet different requirements. The DCE security server uses special and somewhat inconvenient measures to protect the keys of principals in the system. DCE CDS directories are replicated but a desktop is not. A spreadsheet names fine-grained objects (its cells) which present unique scaling problems. New naming systems will continue to appear, particularly in applications.

Up to now, there has been no basic and consistent naming interface. Each naming system has its own API, so application programmers must write custom software for each naming system that their applications use. When applications are ported to different systems, they must be modified to use that system's naming interfaces. As new naming systems are introduced, applications that need to use them must be extended.

There has also been no first-class, generic support for composite names. A few distributed systems support composite names—names that span multiple naming systems. This support is limited and specialized. The DCE name /.../ch.hp.com/sec/principal/jsmith is a composite name. ch.hp.com is resolved in the Internet DNS[3,4] namespace, sec is resolved in the DCE CDS namespace, and principal/jsmith is resolved in the security service's namespace. In DCE only the security, file system, and DCE daemon namespaces can be accessed through composite names. UNIX® rcp uses composite names in a different way from DCE. For example, ajax:/usr/jsmith/naming/memo.txt is an rcp name with two components: ajax is a host name and /usr/jsmith/naming/memo.txt names a file on ajax. DCE and rcp use their own syntaxes and conventions for their names.

Another area of inconsistency between naming systems is their policies for how the namespace is structured. Many systems have very little policy and what policy there is has evolved in a haphazard way. Application writers who use the namespace to advertise their services must follow different conventions for the various environments in which their programs will run or they must invent their own policies for using the namespace. Administrators who configure a site are also faced with confusing, inconsistent, or no policy for how to use the namespace. End users need intuitive ways of finding and naming objects.

Overview of X/Open Federated Naming (XFN)

Several vendors of distributed computing systems realized that they shared these naming problems. Engineers from Digital, HP, IBM, OSF, SNI, and Sunsoft started work on a naming specification in June 1993. In March 1994 version 1 of the Federated Naming Specification[5] went to X/Open® for fast-track review. The specification achieved preliminary status in July 1994. The multivendor team continued to work on extensions to the specification and on validating it before it became part of the X/Open Common Application Environment (CAE) in 1995.

The XFN specification defines application programming interfaces (APIs) and corresponding remote procedure call (RPC) interfaces. XFN specifies a naming syntax for composite names and provides operations to join different naming systems together into a relatively seamless naming federation. XFN also specifies some naming policy.

Fig. 1 illustrates an XFN configuration. The XFN API is layered over a framework into which different context implementations are inserted. A specific context implementation is required for each naming system in a federation. A context implementation maps XFN operations into operations that are native to its naming system. For example, the NIS+[6] context implementation maps operations in the XFN API to corresponding operations in the NIS+ API. A naming system's software below the context implementation is not changed.

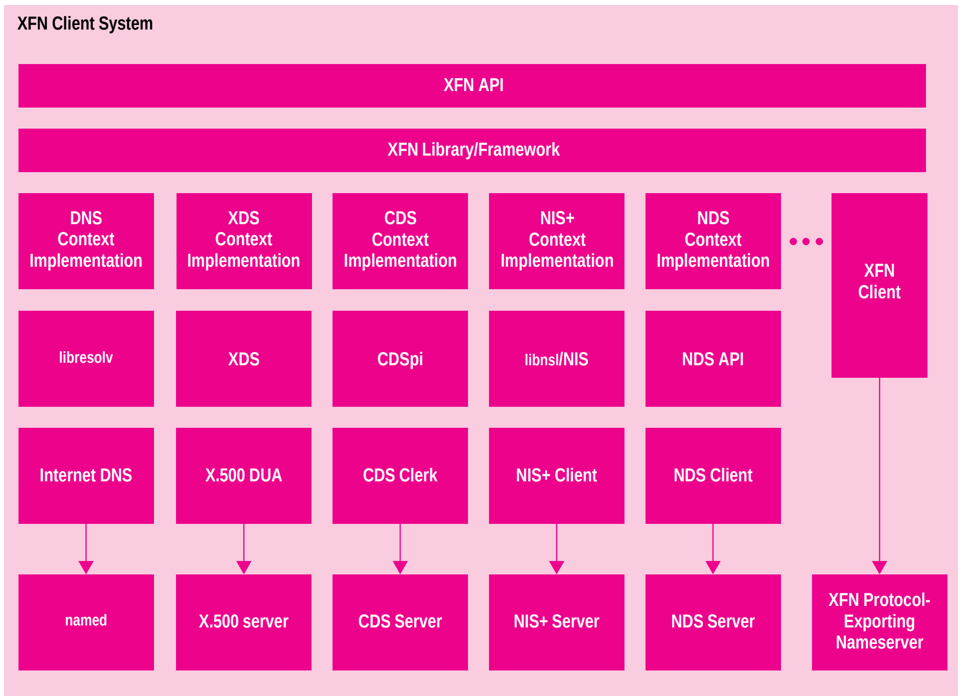


Fig. 1. XFN configuration using client context implementations. A program seeking internal location information for a human-readable name passes the name to the XFN API. The name is broken apart and processed by the appropriate naming systems, and the desired location information is returned by the naming system servers (bottom row).

To join a federation, a naming system must simply provide its specific context implementation.

In Fig. 1 the client-side software for five naming systems runs on the XFN client system. In addition, an XFN client module that imports an XFN protocol is on this system. The XFN client module may do caching and other typical naming client jobs. Each naming client on the system uses its native protocol to communicate with its server.

Definitions

In this section and hereafter in this article, paragraphs in quotation marks are taken directly from the X/Open Federated Naming Specification.[5]

"Every name is generated by a set of syntactic rules called a *naming convention*. An *atomic name* is an indivisible component of a name, as defined by the naming convention. A *compound name* represents a sequence of one or more atomic names composed according to the naming convention."

Case sensitivity, the choice of escape, quote, and delimiter characters, and the order of atomic names in a compound name are common features of a naming convention.

"In UNIX pathnames, atomic names are ordered left to right, and are delimited by slash (/) characters. The UNIX pathname usr/local/bin is a compound name representing the sequence of atomic names, usr, local, and bin. In names from the Internet DNS, atomic names are ordered from right to left, and are delimited by dot (.) characters. Thus, the DNS name sales.Wiz.com is a compound name representing the sequence of atomic names com, Wiz, sales."

"The *reference* of an object contains one or more communication endpoints (addresses). The association of an atomic name with an object reference is called a *binding*. A *context* is an object whose state is a set of bindings. Every context has an associated naming convention."

A UNIX directory is a type of context. An atomic name in one context can be bound to a reference to another naming context object, called a *subcontext*.

"A *naming system* is a connected set of contexts of the same type (having the same naming convention) and providing the same set of operations with identical semantics. In the UNIX operating system, for example, the set of directories in a given file system (and the naming operations on directories) constitute a naming system. A *naming service* is the service offered by a naming system. It is accessed using its interface. A *namespace* is the set of all names in a naming system."

The XFN API

XFN defines uniform naming interfaces that support basic naming functionality. As illustrated in Fig. 1, the XFN interface is layered over specific naming services' APIs. The details of the underlying naming system are hidden from the application. Applications that use the XFN API can access a variety of current and future naming systems without modification.

The operations in the XFN interface range from simple to complex. Simple naming systems are not expected to support the more complicated operations, but the functionality offered by sophisticated naming systems can still be accessed via the XFN API.

The XFN base context interface includes operations to bind an atomic name in a context to an XFN reference and to unbind a name. Other operations in the XFN base context interface look up a name and return its reference, look up a link, list all names and bindings in a context, and create a subcontext.

XFN supports the notion of attributes (or properties) associated with a name. Attributes can be used to provide summary characteristics about the object being named. For example, a printer might be named /.../Wiz.com/eng/os/service /prntr1. The name prntr1 would be bound to an XFN reference that contains the address of the server for that printer. Attributes could also be associated with the name prntr1 that describe its type (LaserJet, inkjet, etc.) and the formats it supports.

Attributes are accessed through the XFN attribute interface, which includes operations to set, modify, and get attributes associated with a name in a context. An attribute consists of an identifier, a syntax, and one or more values. Operations to search for names whose attribute values match a filter expression are also defined. In the printer example, a search operation could be used to locate a LaserJet printer in the eng/os department that supports the PostScript™ format.

The XFN API has been mapped to Internet DNS, CCITT X.500, DCE CDS, and ONC NIS+. Since X.500 provides the most functionality of these naming systems through its XDS/XOM API, this naming system presented the most challenges for XFN. XFN captures the functionality of XDS/XOM but is a simpler, more intuitive API.

Support for Composite Names

XFN specifies a syntax and parsing rules for composite names. Operations to manipulate these names are also provided.

"A *composite name* consists of an ordered list of zero or more components. Each component is a string name from the namespace of a single naming system and uses the naming syntax of that naming system. A component may be an atomic name or a compound name from that namespace." The string form of a composite name is "the concatenation of the components from left-to-right with the XFN component separator (/) between each component."

In the DCE composite name /.../ch.hp.com/sec/principal/jsmith mentioned earlier, the ch.hp.com component is a compound name in the DNS naming system, whose syntax is right-to-left '.' separated. The second component, sec, is in the DCE CDS naming system, whose syntax is left-to-right '/' separated, like XFN's syntax. The final two components, principal/jsmith, are in the DCE security naming system. This naming system's syntax is also left-to-right '/' separated. Since a component is defined as the name between two XFN separators, principal/jsmith is two components even though both are in the same naming system.

Composite names are formed when naming systems are joined by binding location information about a context in one naming system into its parent context in another naming system. This location information about a context in another naming system is called a *next-naming-system pointer*. For most naming systems a next-naming-system pointer is bound to a leaf name in its namespace and is treated like any other name in its namespace. The location information is represented in an XFN reference. The XFN bind operation can be used to create next-naming-system pointers.

Fig. 2 shows how the name /.../Wiz.com/user/jsmith/fs/naming/memo.txt is composed. /... is a reserved token that indicates the root of a global naming system. The Wiz.com component is a name in the DNS naming system, user/jsmith/fs is in the DCE CDS naming system, and naming/memo.txt names a DFS file. Location

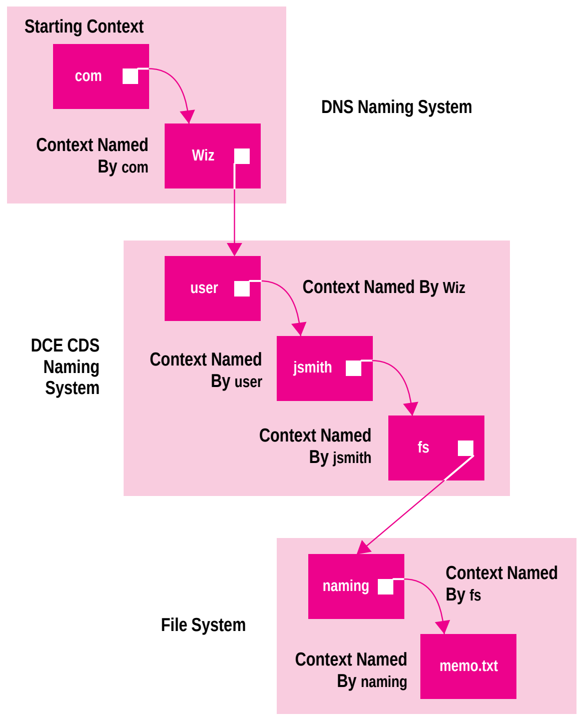


Fig. 2. Next-naming-system pointers (Wiz and fs).

information about the DCE CDS context in which user is bound is associated with the name Wiz in DNS. The atomic name fs is bound in the CDS context user/jsmith to an object reference with location information of jsmith's home directory in DFS. Wiz and fs are next-naming-system pointers.

The XFN framework controls path resolution of a composite name. To resolve /.../ Wiz.com/user/jsmith/fs/naming/memo.txt the XFN framework first invokes the DNS context implementation to resolve Wiz.com. The DNS context implementation makes libresolve calls to gather the information it needs to form the XFN reference associated with Wiz.com, which it returns to the framework. The framework inspects the reference and invokes the context implementation specified in the reference. The framework passes to the context implementation the location of the starting context for resolution and the remaining components to be resolved. In this example, the context implementation is for DCE CDS, the starting context is the one named by Wiz.com, and the name to be resolved is user/jsmith/fs/naming/memo.txt. CDS can only resolve user/jsmith/fs. It returns the XFN reference bound to user/jsmith/fs and the remaining components to be resolved back to the framework. The framework then passes the remaining name, naming/memo.txt, to the file system to complete the resolution.

XFN Protocols and Configurations

XFN specifies client-server RPC interfaces for use with two RPC protocols: DCE RPC and ONC RPC. The protocols support the operations in the XFN API. New naming systems and some current ones are expected to use one of these protocols for their client/server communications.

"The advantage for naming systems that export an XFN protocol is that any existing XFN client that imports the protocol can be used to communicate with it. This is particularly useful for applications that need to export naming interfaces. Application programmers do not have to duplicate the client-side implementation and they do not have to invent new naming interfaces. This provides additional benefits such as the ability to use caching and other mechanisms provided by the XFN client implementations, and a direct (and possibly more efficient) mapping of XFN operations to the naming operations."

The XFN naming model presents a hierarchical namespace that incorporates different naming systems. The naming systems are connected together into three levels. The top level is a global namespace; X.500 and DNS are expected to control this level. The next level is an enterprise namespace; DCE CDS, ONC NIS+, Banyan Streettalk, and Novell NDS[7] are considered enterprise naming systems. The third level is the application namespace. The DCE security service, a file system, and a desktop support application namespaces.

The XFN model, API, and protocols provide a toolkit for configuring naming federations in various ways. Fig. 1 illustrates a heavyweight XFN client system with context implementations and client-side code for five naming systems and a module that imports an XFN protocol. Fig. 3 shows a lightweight XFN client system that only runs the naming module that imports an XFN protocol. Multiple name servers export the XFN protocol. Two of the name servers use a variation of their context implementations to map arriving XFN calls to their naming systems' native operations. These servers also export their native protocols to support clients running

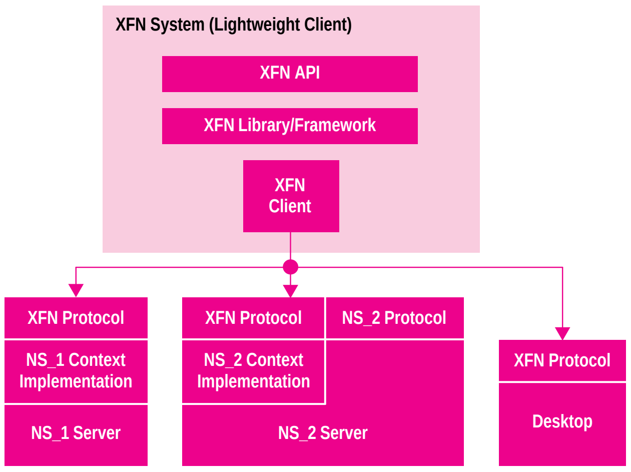


Fig. 3. Lightweight XFN client configuration with multiple name servers.

legacy software. The desktop application was originally written to export its namespace with the XFN protocol.

The two systems shown in Fig. 4 are a lightweight XFN client and a server that acts as an intermediary. Like the client in Fig. 3, the XFN client in Fig. 4 only runs the naming module that imports an XFN protocol. None of the legacy systems' client-side software needs to run on this system. Depending on the client system's requirements, the XFN client can be implemented and configured to consume more or less resources. For example, the XFN client might defer to the caching mechanisms provided by the native naming

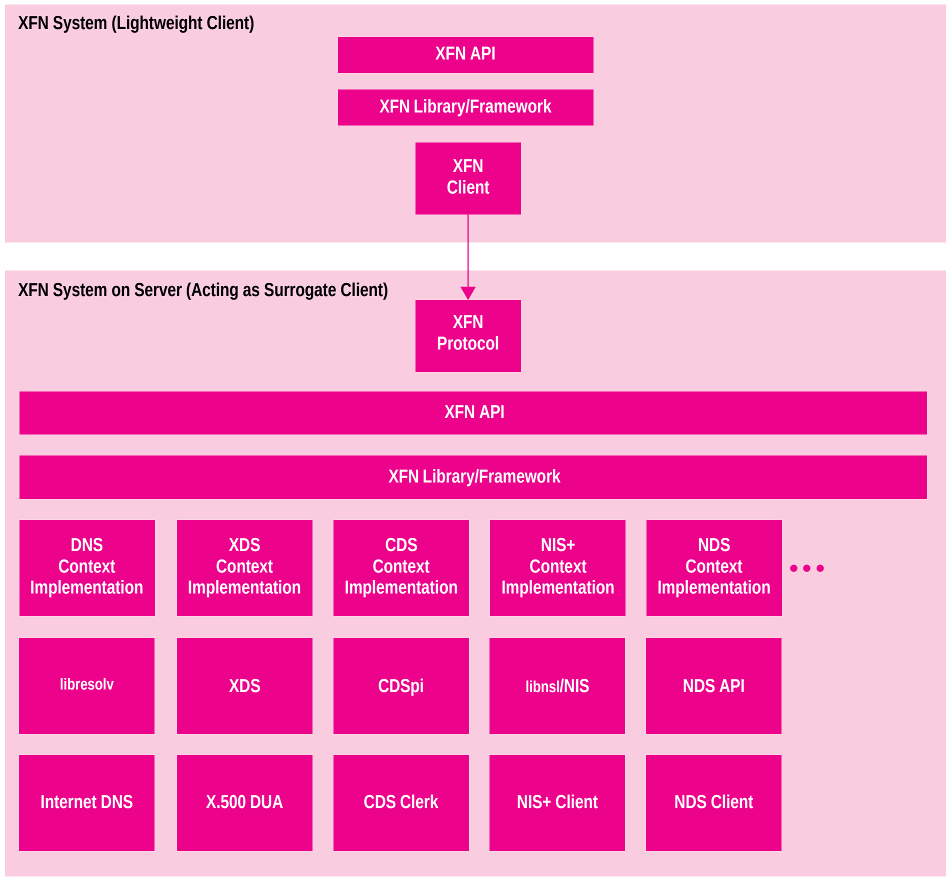

XFN System (Lightweight Client)
XFN API
XFN Library/Framework
XFN Client
XFN System on Server (Acting as Surrogate Client)
XFN Protocol
XFN API
XFN Library/Framework
DNS Context Implementation
XDS Context Implementation
CDS Context Implementation
NIS+ Context Implementation
NDS Context Implementation
• • •
libresolv
XDS
CDSpi
libnsl/NIS
NDS API
Internet DNS
X.500 DUA
CDS Clerk
NIS+ Client
NDS Client

Fig. 4. Lightweight XFN client configuration with surrogate client on server.

system clients. "The legacy naming system clients in Fig. 4 reside on a remote system (similar to Fig. 1) that also exports the DCE XFN protocol. This remote client can be viewed as a surrogate or proxy client that acts on behalf of the initial requestor and performs the native naming system functions."

Another common XFN configuration combines Figs. 3 and 4. Some name servers export the XFN protocol and can be accessed directly from the lightweight XFN client. Other name systems are accessed via an XFN surrogate client.

XFN Enterprise Policies

The three-level hierarchy of global, enterprise, and application namespaces is an XFN policy that was mentioned in an earlier section. Major entities, such as countries and organizations, are named in the global namespace. Names in a global naming system change infrequently and require sanction from a global authority to do so. The enterprise namespace is assumed to contain names that are local to an organization. XFN policies provide some guidelines for structuring an enterprise namespace. These policies do not apply to the global or application namespaces.

XFN policy recognizes that there are commonly named objects in an enterprise. These are organizational units, users, hosts, services, and files. XFN policy reserves tokens to identify namespaces for these objects and also applies a relationship between them. Table I summarizes XFN enterprise policies. Some examples of names that use XFN policies are:

- /.../Wiz.com/_orgunit/r-d/eng/os/_user/jsmith/_fs/naming/memo.txt. Names jsmith's file naming/memo.txt. jsmith is a user in the r-d/eng/os department of the Wiz.com company.
- /.../Wiz.com/_orgunit/sales/_user/mjones/_service/calendar. Names the calendar service for mjones who is a user in the sales department of the Wiz.com company.
- /.../Wiz.com/_orgunit/newton/bldg300/conf-rm/chaos/_service/calendar. Names the calendar service for the Chaos conference room in building 300 of the Newton site of the Wiz.com company.

Programs that use XFN policies are more portable across computing environments and enterprises. A distributed application, such as a calendar service, has a standard place (a _service context) to put its binding information. An administrator can put information about each user and each host in a central, predictable place. An end user can more easily figure out how to name another user's files, for example.

Despite the fact that XFN policies are minimal, they are controversial. Standard token names raise concerns of name collisions. XFN specifies these tokens on the premise that the benefits of a more structured namespace outweigh the risk that XFN tokens will collide with names that are already in a namespace. An XFN implementation can sacrifice its portability and customize its own tokens to identify the namespaces for common objects. An XFN implementation can conform to the XFN API but to some or none of the XFN policies. An enterprise namespace will normally have many contexts that are outside of the XFN policy domain and may have additional policies of its own.

Table I
XFN Enterprise Policies

Context Type	Context Type Token	Parent Context	Subordinate Context
Organizational Unit	_orgunit	enterprise root	user, host, file system, service
User	_user	enterprise root, organizational unit	service, file system
Host	_host	enterprise root, organizational unit	service, file system
Service	_service	enterprise root, organizational unit, user, host	not specified
File System	_fs	enterprise root, organizational unit, user, host	not specified

Other Naming APIs

Some naming APIs, such as the DCE RPC Name Service Independent (NSI) Interface[8] and the OMG Common Object Service's Naming Service Interface,[9] are customized interfaces that may be layered over an XFN API and its implementation.

The RPC NSI provides a high level of abstraction for navigating a namespace and yielding DCE RPC location information in the form of RPC binding handles. The OMG naming interface is a subset of the XFN basic context interface. The OMG interface maps names to CORBA object references. Unlike RPC NSI and the OMG naming interface, XFN accepts many different types of object references and provides mechanisms to extend the set of object references. Also, neither the DCE RPC NSI nor the OMG naming interface has support for attributes.

When these customized interfaces are implemented over XFN, they take advantage of XFN benefits such as portability and federation and they leverage all the software that supports the XFN API.

Conclusions

Among the benefits that XFN provides are:

- A uniform naming interface for accessing different naming systems.
- Application programming interfaces as well as RPC interfaces.
- A naming syntax for composite names.
- Operations to join different naming systems together into a naming federation.
- A framework that supports the addition of new naming systems to an XFN federation with no changes to applications or to current member naming systems. A naming system that joins a federation must only supply a context implementation that maps the XFN API or an XFN protocol to its native

operations. Otherwise, the naming system's software is not changed.

- Support for small clients.
- Easier administration of the various naming systems in a distributed computing environment. Browsers and editors that are written to the XFN API can access an entire federated namespace.
- Application power. XFN applications can access a wide variety of naming systems through the same simple, yet functional API.

Future Directions

Future work needs to be done on policy. Different vendors that offer similar applications need guidelines for sorting out their uses of the namespace. Users sometimes want to select among similar or replicated services based on network topology or load balance. Administrators often have common information about a group of users and customized per-user information. Namespace policies and software could support these requirements.

Acknowledgments

This paper is a summary of the X/Open Federated Naming Specification. Quoted paragraphs are taken directly from the specification as are some of the figures and tables. The X/Open Federated Naming architecture team includes: Rangaswamy Vasudevan, Rosanna Lee, and Vinnie Ryan from Sunsoft, Ellen Stokes and Dave Bachmann from IBM, Norbert Lesser and Arthur Harvey from OSF and the author from HP. Joseph Pato from HP, Arthur Gaylord from the University of Massachusetts at Amherst, and Richard Curtis from Banyan were early reviewers and are consultants to the architecture team. Peter Dejong, Larry Derany, Michael Kong, and Joseph Pato provided valuable review comments.

References

1. *Information Technology—Open Systems Interconnect—The Directory*, CCITT X.500 (1988, 1993)/ISO Directory, ISO/IEC 9594: 1988, 1993.

2. *X/Open DCE: Directory Services*, X/Open Preliminary Specification, December 1993.

3. P.V. Mockapetris, *Domain Names—Concepts and Facilities*, Internet RFC 1034, November 1987.

4. P.V. Mockapetris, *Domain Names—Implementation and Specification*, Internet RFC 1035, November 1987.

5. *Federated Naming: The XFN Specification*, X/Open Preliminary Specification, July 1994.

6. R.Ramsey, *All About Administering NIS+*, SunSoft Press.

7. D. Bierer, et al, *Netware 4 for Professionals*, New Riders Publishing, 1993.

8. *X/Open DCE: Remote Procedure Call*, X/Open Preliminary Specification, October 1993. Specifies RPC NSI.

9. "Naming Service Specification," *OMG Common Object Services Specification, Volume 1*, March 1994.

HP Integrated Login

HP Integrated Login coordinates the use of security systems and improves the usability of computer systems running the HP-UX* operating system.

by Jane B. Marcus, Navaneet Kumar, and Lawrence J. Rose

The HP Integrated Login product provides major usability gains for customers deploying enhanced security technologies on computer systems based on the HP-UX operating system. In this article, we describe the customer needs and the HP Integrated Login solution.

As computer networks expand, and as pirates more frequently travel the information superhighway, customers require more stringent methods for securing data and accounts. The base HP-UX operating system provides standard UNIX® security mechanisms, but these do not meet all the needs of security-minded customers. There are many security technologies available commercially and in the public domain. HP customers sometimes wish to deploy one or more of these technologies on the HP-UX platform.

Security technologies use passwords to verify the user's identity and determine access rights to data and services. A user must enter a password and the password must be verified before access is granted. For example, basic HP-UX security requires that a password be entered for the user to gain access to the HP-UX machine. In addition to machine entitlement, passwords also may be used to verify the user's right to access protected services (e.g., mail systems) in the user's environment.

Security-minded customers see many benefits to deployment of enhanced security technologies—for example, protection against impostors and network eavesdroppers. However, placing additional security technologies on top of the HP-UX system can create a burden to the users of the system. When multiple security technologies are deployed (to monitor access to various protected services in the user environment), each technology requires password verification. Thus, a user may be forced to type in a password for the HP-UX system and then for each additional security technology. Furthermore, the use of multiple security technologies creates a complex task for users when passwords need to be changed in multiple places.

Customers need enhanced security, but they also want usable systems. Customers want to operate in a familiar environment, and do not want to learn many new commands for accomplishing basic tasks. When faced with a lengthy or complicated process, typical users may ultimately compromise the security of their systems by writing down passwords and procedures that might otherwise be forgotten. Customers will not accept a burdensome process for their users.

HP Integrated Login

The HP Integrated Login product has evolved to meet the customer needs discussed above. The original product for the HP-UX 9.x operating system was developed in response to DCE† customer requirements and was delivered primarily for use by HP's DCE customers. However, with the HP-UX 10.0 release, the HP Integrated Login product has been made extensible, so that it can serve the HP-UX community at large. The latest HP Integrated Login provides library interfaces that allow a generic set of security technologies to be integrated with HP-UX. The customer has maximum flexibility to choose and deploy appropriate technologies. Since DCE has an outstanding security technology, we expect that HP Integrated Login users will most often choose DCE for their security needs, but the HP Integrated Login product can support other technologies equally well.

The primary purpose of the HP Integrated Login product is to allow HP-UX users a convenient method for incorporating other security technologies into the standard HP-UX environment. Users should be able to use familiar HP-UX tools to accomplish familiar tasks. Thus, HP Integrated Login extensions have been added to several standard HP-UX 10.0 utilities.

The most important functionality delivered by HP Integrated Login is a single-step login: the user enters a password once at login time, and this password is used to grant access to the HP-UX machine as well as verify access among all the configured security technologies. The HP-UX 10.0 commands `login` and `su` have been enhanced to include single-step login capabilities. Also, the HP user desktop (HP VUE) has been integrated to support multiple security technologies. Login information is propagated throughout the entire VUE session and logins need not be repeated when new VUE windows are opened.

Password consistency is fundamental to most HP Integrated Login deployments. A user chooses one password, and this password is adopted across all security technologies. Thus, the user can supply the password once and the HP Integrated Login utilities transparently perform logins to each configured security technology on behalf of the user. The HP-UX 10.0 `passwd` command has been integrated to synchronize passwords for the user, so that a requested password change can be propagated to all configured security technologies. Likewise, user information commands `chfn` and

† DCE is the Distributed Computing Environment. See article, page 6.

chsh are provided to allow changes to finger and user shell information across security technologies. (Finger information includes the user's real name, location, and telephone number.)

It is typical of the UNIX operating system that several operations are password-controlled—for example, file transfer to or from a remote machine and screen lock or unlock. Integrated file transfer protocol (ftp) and HP VUE lock utilities have been provided. Consistent with other HP Integrated Login utilities, these operations verify user access for all configured security technologies based on one user-supplied password.

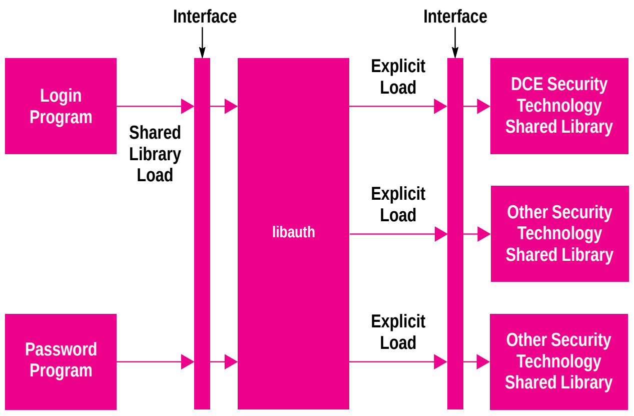
The HP Integrated Login product provides extensions to HP-UX commands to support multiple security technologies on top of the HP-UX system. The extension method involves a new shared library provided with HP-UX 10.0. Integrated HP-UX utilities make calls to this shared library (libauth.sl). The libauth library calls handle various security tasks such as password verification for login and password changes. Thus, HP-UX utilities relinquish to libauth the direct responsibility for supporting diverse security technologies. Furthermore, these HP-UX utilities have no awareness of multiple security technology configurations, and have no knowledge of the details of how these security technologies function.

HP Internal Customer Needs

Enhanced security technologies on top of HP-UX have existed for some time. Before the creation of HP Integrated Login, several security technologies had independently been integrated into the HP-UX system. Each security technology had its own login method, and each security product would spin off new versions of HP-UX login commands and HP VUE to incorporate the technology's login implementation. These efforts were difficult to coordinate, and there grew to be many different versions of HP-UX login commands to accommodate all of these security technologies. One HP Integrated Login goal was to replace the myriad of login implementations with one generic login methodology. This was expected to solve a number of different problems, including the HP support cost to maintain multiple code bases. The solution required the definition of a generic login procedure, flexible enough to accommodate all the existing login methods.

Extensibility

HP Integrated Login supports multiple security technologies. The HP Integrated Login configuration file declares which technologies are being integrated. Typically, the HP-UX machine administrator uses HP Integrated Login administrative tools to create and maintain this configuration file. Each technology declared in HP Integrated Login's configuration file must provide a technology shared library. This shared library will be dynamically loaded by the HP Integrated Login library (libauth), which coordinates all underlying security technologies. The libauth library determines the names of the technology shared libraries and the order in which to load them based on the contents of the HP Integrated Login configuration file. Fig. 1 shows the resulting architecture. The libauth library needs no special knowledge of any security technology library that it loads. Well-defined interfaces exist between the libauth coordination library and the security technology shared library.



Interface
Interface
Login Program
Shared Library Load
Password Program
libauth
Explicit Load
Explicit Load
Explicit Load
DCE Security Technology Shared Library
Other Security Technology Shared Library
Other Security Technology Shared Library

Fig. 1. The extensible architecture of HP Integrated Login.

From HP-UX commands, the following can occur:

- The HP-UX command dynamically loads the HP Integrated Login shared library (libauth).
- The libauth library reads the HP Integrated Login configuration file and dynamically loads the configured security technology libraries.
- The HP-UX command makes library calls to libauth to handle login and password functions.
- The libauth library makes library calls to security technology libraries.

An exception in the HP Integrated Login library strategy is the method by which basic HP-UX security is provided. While the HP Integrated Login configuration may specify the use of basic HP-UX security, there is no HP-UX technology library to be dynamically loaded. Rather, the HP-UX commands handle basic HP-UX functions from within the command code.

The libauth library is shipped with the HP Integrated Login product, and is dynamically loaded by the integrated HP-UX commands and HP VUE. HP-UX utilities use libauth to integrate security technologies, but the basic HP-UX security code is always accessible since it is contained within the HP-UX utilities. Thus the HP-UX 10.0 utilities can still provide standard HP-UX functionality on systems that do not have libauth installed. While the integrated commands must be aware of their use of libauth, the commands are completely unaware of libauth's use of underlying security technology libraries.

Configuration of Multiple Technologies

The HP Integrated Login configuration file is used to define a login policy to be used on a particular machine. The policy specifies which technologies are in use. When multiple security technologies are in use, the relative priority of these technologies must be configured for HP Integrated Login operation. One technology is configured as the primary login technology. This primary login technology will be the initial technology to be consulted for user password verification. If the primary login succeeds, the user will be granted access to the HP-UX machine, and additional logins can then proceed (transparently) to verify the user with other security technologies.

In case the primary login does not succeed, a fallback technology can be configured. If the user can be verified with the fallback security technology, the user is granted access to the HP-UX machine, and again, other configured logins can then proceed.

The importance of the fallback strategy cannot be understated. Security technologies often have dependencies on network communications and cannot function if the network is not intact. For some customers, it is unacceptable for users to be denied access to their local machines because of network problems. The HP Integrated Login fallback strategy allows customers who require a high level of robustness to use HP products with confidence.

The relationship between the primary login technology and the fallback login technology must be well-understood. In some cases, the primary login technology may attempt to synchronize the fallback technology with current user information. For example, when DCE serves as the primary login technology, it is an HP Integrated Login option to automatically populate the HP-UX user information database (i.e., the /etc/passwd file) with information from the DCE security database. In most cases, the /etc/passwd file will never be accessed at login time, because DCE, as the primary login technology, will verify the user. However, when DCE is unavailable, HP-UX login security can be used as a fallback to log in all users known to DCE. Such an arrangement is advantageous for administrators who want to maintain user accounts in one primary location, but also want to facilitate fallback logins where necessary.

In other cases, administrators may purposely wish to maintain some users in one security technology database and other users in a different security technology database. The HP Integrated Login configuration of primary and fallback login technologies can facilitate this process. The HP Integrated Login libauth library will consult the primary login technology first to verify the user, but if this user is not known to the technology, HP Integrated Login can be configured to consult the fallback login technology.

In addition to configured primary and fallback login technologies, other login technologies can be configured. These logins will be done transparently for the user, in the order in which they have been configured with HP Integrated Login. The purpose of these additional logins may be to enable user access to some protected service in the user's environment. These additional logins will only be attempted if the primary or fallback login has succeeded, that is, if this user has been granted access to the HP-UX machine. Errors occurring with these additional logins are nonfatal, meaning that the user session can proceed even if one or more of these additional logins fails.

The organization of technologies in the configuration file is used by libauth to determine the order in which technologies should be loaded and accessed. We call libauth's ordering of technologies the *policy chain*, and it reflects the configuration of primary login, fallback, and additional login technologies. The policy chain is used by libauth to make decisions on how to sequence through the configured technologies.

The configuration file may also contain directives to libauth regarding handling of login errors. These directives logically become part of libauth's policy chain and determine libauth's actions in the event of login failures. For instance, the configuration file may specify that a login failure because of an incorrect password entry should result in a denial of machine access, regardless of whether this password may be verifiable by other configured technologies in the policy chain. The libauth library's actions in this case would be to stop the login sequence after the initial failure and refrain from cycling through the security technologies. The behavior of libauth is configurable, so it is also possible to specify a configuration that authorizes libauth to pass the login request to the next technology in the policy chain.

The configuration file also includes a mechanism to pass configuration information to the security technology libraries that will be loaded by libauth. Configurable parameters can be specified for each specific security technology. These parameters are meaningful only to the security technology library and are determined by the security technology library provider. For instance, a specific security technology may support the notion of a session lifetime. A configurable parameter called LIFETIME may exist in the HP Integrated Login configuration file to be passed to the security technology when being loaded by libauth. The libauth library will pass the configuration information to the security technology library, but will not use or process this information in any way (thus preserving the extensibility model).

libauth Login Processing

To accomplish a login that results in a user session, several behind-the-scenes events must occur. The procedure consists of three phases: the initialization phase, the login phase, and the session-setup phase.

During the initialization phase, the HP Integrated Login policy is read from the configuration file. The libauth library proceeds to load the security technology libraries and charges them to run through their respective initializations. Initialization failures from any of the security technology libraries cause libauth to mark the technology as inaccessible. When security technologies are inaccessible, libauth must adjust its understanding of the policy chain to reflect the effective policy. Upon successful initialization, libauth and the security technology libraries exchange entry point information. This makes it possible for two-way communication to occur between the security technology library and libauth. The libauth library can now call the security technology library interfaces to handle security tasks, and the security technology libraries can communicate messages and error status.

The login phase is a two-step process. Step one determines whether the user should be granted access to the local system. Prompts are issued for the user name and password, and subsequently the primary login technology library is called to verify access. Logically, this step may be considered a Boolean operation which simply returns a yes or no answer regarding the user's entitlement to access the local system. Depending upon the configured policy chain, libauth may continue on failure to the configured fallback technology, or may deny access.

The second step in this login phase (after having granted machine access) is to complete any additional logins that should be done. These additional logins may be needed to enable operations with some protected services once the user session begins. In current implementations, additional logins can only succeed if the password entered is valid for this user across all security technologies. However, libauth code is in place to support different passwords for additional security technologies, although this code is not yet in practical use. The method for supporting multiple passwords depends on the primary login technology's ability to securely store passwords to be used with other security technologies. A successful login with the primary login technology would result in the stored passwords being passed to libauth for use with the other technologies in the policy chain.

The session-setup phase is for establishing the user session. The information about how to set up the session is retrieved from the underlying primary login technology database. In particular, the user and group IDs must be set for the new session, and the user shell must be started. In addition, the exportation of environment variables occurs. If any configured technologies require special environment variables to be set, these environment strings are passed back to the HP-UX command so that they are exported at session-setup time.

Password and Information Processing

The libauth library interfaces oversee changes to user information, such as password, user shell, and finger information. The HP-UX passwd command, for example, loads libauth to coordinate password changes. The libauth (and technology library) initialization phase described for login processing is the first step here as well.

Before calling libauth to make a password change, the passwd command calls libauth to check the new password that has been proposed. Most security technologies apply password strength checking algorithms to newly created user passwords. These algorithms test whether the new password meets certain criteria. For example, one HP-UX requirement is that the new password must have at least two alphabetic characters and at least one numeric or special character. For password strength checking, a libauth interface determines if the selected password is acceptable to all configured security technologies. The password is rejected if any of the security technology libraries rejects it, and the operation fails.

If the proposed password is acceptable, the command calls libauth to contact the primary login technology. The primary login technology will then change the password in the primary login technology's user database. Failure to change information correctly with the primary login technology causes the entire operation to fail. If the change succeeds, libauth follows the policy chain to request password changes in all other security technology databases. If a failure occurs for this user with any of the additional (i.e., optional) technologies, an error indication is recorded and the next technology in the chain is tried. If a failure occurs during a password change operation, the password may no longer be consistent across all technologies. An error indication clearly states this and gives advice on how to remedy the situation manually.

Some details of the policy chain configured in the HP Integrated Login configuration file do not apply to password processing. The configuration file is used to determine the primary login technology. However, libauth password interfaces make no attempt to deal with a configured fallback technology in case of error.

The libauth library interfaces allow other user information to be changed. The user's shell can be also changed. In all cases, changes to user information start by attempting to make the change in the database of the primary login technology. Successful changes cause the operation to continue down the policy chain to completion.

Other libauth Interfaces

Aside from password and login functionality, other libauth interfaces are available to HP-UX commands.

For security technologies that include the concept of login expirations, libauth supports a refresh operation. For example, suppose that the user's machine has been left locked by the VUE screen lock program, and the user's login expires before the user returns to unlock the machine. The libauth refresh interface allows bypassing some of the details of the full login process, although reprompting of the password is required for this step to maintain security.

Another libauth interface resets the current login information. This interface is used by commands that can switch between users, such as ftp and su. The reset action cleans up any residual login information, effectively terminating a previous login across all security technologies.

Choosing a Primary Security Technology

An HP Integrated Login goal is to simplify user administration among multiple security technologies. When multiple security technologies are deployed, multiple user information databases may coexist. These *registries* are repositories of user-related information, and different technologies require the storage of different types of information about a user. HP Integrated Login configuration of a primary login technology determines which registry is most important for maintaining user information. The primary login technology's registry assumes the role of the main location for user information, and registries from other technologies are logically subservient. For example, if the password that the user has provided is determined to be incorrect when checked against the main registry, HP Integrated Login may be configured to deny access without further checking of the password against other technology registries. If the user requests a change of password and for some reason the main registry cannot entertain the change, no attempt is made to request the change in other registries.

When deploying multiple security technologies, the choice of the main registry is very significant. System administrators might ask what features such a registry should have. Especially in a networked environment, this registry must be highly available and reliable, since users may be denied access if it is not in operation. The main registry must be capable of storing critical user information, including but not limited to user name, user identifier, password, group identifier, home directory, and login shell. Since user information is likely to change as we move towards more complex systems, built-in extensibility of the registry is highly desirable.

We find DCE to be an excellent choice for the main security technology. The DCE security service registry satisfies all basic requirements for a main registry.

The DCE registry is highly available. The implementation allows for a collection of one master and several replicas, so user information can be obtained from multiple (but consistent) sources. Replicas are copies of the master information and are read-only sources of information. If a replica goes out of service, others are available to provide user information. If the master goes down, a suitable replica can be transformed into a master. If service degrades because of heavy demand, additional replicas can be added to expedite requests. Consequently, the service is scalable and reliable.

DCE uses Kerberos™ authentication protocols and is highly secure in a distributed environment. For example, DCE does not transmit passwords in the clear across the network. This feature is not particularly useful if other technologies do otherwise, but the main registry should not be the weak link.

The OSF DCE 1.1 registry is extensible. System administrators can extend the registry to hold arbitrary user information. For example, DCE 1.1 uses this extensibility feature to support password aging (mandating that passwords be changed regularly) and password strength checking.

DCE is serviceable. Logging and audit trails can be used to diagnose error conditions.

In summary, we find DCE to be the logical choice for the primary login technology and to serve as the main registry of user information. (See the article on page 41 for more information about DCE security services.)

Login Access Using HP Integrated Login and DCE

Although the HP Integrated Login design does not require it, our implementation works very well with DCE providing the main registry of user information. This solution is robust and allows administrators to focus their efforts on maintaining user information in one location: the DCE registry. Fig. 2 illustrates how HP Integrated Login uses the DCE registry.

Customers with robustness requirements often choose to configure HP-UX security as a fallback technology. As described earlier, the fallback technology is used when the primary login technology is unavailable. With DCE as the primary login technology, DCE is unavailable when the network is not operational. Since HP-UX security is local to the machine and is unaffected by network errors, an HP-UX fallback may be a good choice. However, if the fallback registry is to provide access that is consistent with the main registry, it must be kept consistent with the main registry. Support for keeping the registries synchronized is obviously needed.

For this purpose, DCE provides a tool, passwd_export, that exports the information in the DCE registry to the native HP-UX registries, /etc/passwd and /etc/groups. When HP Integrated Login is installed, a system administrator can configure the HP-UX command cron to run passwd_export periodically to keep the registries consistent.

DCE user accounts are valid across the DCE network environment. Once established in the DCE registry, a DCE user can log in from any machine in the user's DCE environment, with no other special administration required. The DCE registry is implemented as a centralized network service, with requests travelling back and forth over the network from the registry to the client machines. However, DCE allows registry user information to be overridden at the local machine level. In this case, information is taken from the local machine rather than from the DCE network registry. An override mechanism is important to machine administrators wishing to customize their individual machines. For example, in a traditional UNIX system, each machine has a superuser account root. Machine administrators do not want the root account to share a password with all root accounts on machines in the DCE networked environment. Rather, machine administrators want to maintain the password for the root account locally. In this case, administrators must override the information in the main registry in favor of information stored on the local machine.

To handle such cases, DCE provides support for an override file. This file has a format similar to the traditional UNIX /etc/passwd file. If a user is maintained in the override file, the user's access to the machine is verified based on the override file entry and is not verified by the DCE registry. By common use, root is maintained in the override file to ensure that for superuser privileges a local password is required. The DCE override file is only readable by the superuser account and as such is more secure than the HP-UX /etc/passwd file.

Since DCE is a scalable, reliable, and secure service, some installations with especially stringent security requirements may wish to disable fallback login verification. In general, customers can rely on DCE to provide consistent service as

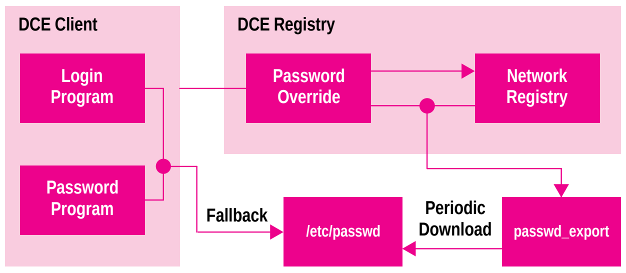


Fig. 2. Integrated login using the DCE registry.

long as the network is operational. Some customers disable fallback to basic HP-UX security because the HP-UX /etc/passwd file is inherently less secure than many customers require. For HP Integrated Login DCE configurations with no fallback technology, most logins will be disabled if there are network problems. However, users being administered in the DCE override file will still have login access in case of network failure, since the override file is stored on the local machine and is unaffected by network errors.

Login Information Maintenance: DCE and HP-UX

Suppose HP Integrated Login has been configured with DCE as the primary login technology and HP-UX security provides the fallback technology.

Example 1. A user wishes to change a password. We have two registries to consider: DCE and HP-UX. The following sequence of events occurs:

- Since DCE is the main registry, the old and the desired new passwords are obtained and passed to the DCE security technology library.
- The DCE registry verifies that the old password was correct and further determines if the new password is strong enough. If these checks pass, the user's password is changed in the DCE registry.
- No attempt is made to contact the fallback registry (/etc/passwd) at this time. However, libauth could propagate the password change to other configured technologies.
- After a certain interval configured by the system administrator, passwd_export runs and exports the changed password information to the HP-UX /etc/ passwd file. Thus, the fallback plan to HP-UX remains intact with this synchronization.

Example 2. A user whose account information is stored in the DCE override file requests a password change:

- The libauth library passes the request to the DCE security technology library. When user account information is kept in the DCE override file, passwords are changed in the override file only. The DCE registry is not changed at all.
- When passwd_export runs, it exports the changed password from the override file to /etc/passwd. This is how the local root user can change its password.

Example 3. A user requests a shell change:

- The chsh command calls libauth to pass the new shell information to the primary login technology library (DCE).
- If configured, passwd_export runs and exports the changed shell information to the HP-UX /etc/passwd file. Thus, HP-UX and DCE registries remain synchronized.

If passwd_export is not run periodically, some traditional UNIX commands and library calls with dependencies on the /etc/passwd file might use stale data. For example, the ls command gets user information from the /etc/passwd file. If the /etc/passwd entry for this user is not kept consistent with the information in the DCE registry, the ls command may be relying on old data for this user.

Login Information Maintenance: NIS and DCE

Suppose HP Integrated Login has been configured with HP-UX as the primary login technology. By way of clarification, it should be noted that NIS support falls under the generic HP-UX umbrella because HP-UX commands have been integrated with NIS for several years. At present, the code to support NIS is retained in the HP-UX commands. Thus, when HP-UX security has been configured, this effectively can include NIS if it has been deployed in the HP-UX environment. Currently there is no way to ensure full account consistency between NIS and DCE because there is no NIS utility comparable to DCE's passwd_export. Thus, users added to the NIS registry must also be added to the DCE registry by the system administrator.

Example 4. A user password change is requested and the HP-UX system (NIS) is the main registry, that is, password verification by NIS determines the user's right to machine access. Suppose also that DCE has been configured for additional login because users need access to some DCE services.

- The user wishing the password change must present the old password and the desired new password. The passwd command calls libauth.
- The libauth library tells the passwd command that the HP-UX system has been configured, so the passwd command inline code handles this password change operation.
- If the user's account is in the local /etc/passwd file, the password is changed there. If the user's account is maintained in the central NIS registry, the password is modified there.
- The passwd command calls libauth to propagate the password change to other configured registries. This request is passed to the DCE technology library and, if successful, results in a password change in the DCE registry. If for some reason the password cannot be changed in the DCE registry, the user is advised to try changing the DCE password again later. The passwd command now provides a command line option to change a password in a specified registry.

The Single Sign-on Problem

HP Integrated Login operates on HP-UX machines only. Much work remains to be done for customers who need a higher level of flexibility and integration. For example, a PC user on a Novell network would like to enter a password at network login time and have this password also validate access for other integrated systems. Unfortunately, there are extremely complex problems associated with login and password synchronization across operating systems and across hardware platforms. This larger problem is often called the single sign-on problem, and is being addressed by an industry working group of which HP is a coleader.[1]

Summary

The HP Integrated Login product addresses the needs of customers wishing to deploy multiple security technologies. HP Integrated Login improves usability by providing single-step login. Options to configure fallback login technologies ensure robustness in the event of network failure. HP Integrated Login is especially convenient for customers deploying DCE, because DCE and HP Integrated Login together provide the tools required for maintaining a high level of consistency between DCE and the HP-UX system for user account information.

HP Integrated Login has been made extensible, beginning with the HP-UX 10.0 release. HP-UX customers are not locked into any particular security technology by design, and customers can incorporate new technologies without changing the underlying commands framework. Customers use the same set of UNIX tools that they are already familiar with, because these utilities now use the HP Integrated Login shared library to support multiple security technologies. Costs to HP are reduced by the centralization of support for multiple security technologies on the HP-UX platform.

Acknowledgments

We wish to acknowledge the efforts of our teammates in creating the HP Integrated Login product. In particular, the contributions of project manager Frederic Gittler and Daniel Nguyen are outstanding. Also, a special mention must be made of John Brezak, who did early prototype work for the HP Integrated Login project.

Reference

1. *OSF Single Sign-On (SSO) Working Group Draft Paper/RFC*, February 1, 1995. Accessible on the Worldwide Web at URL http://www.dstc.qut.edu.au/MSU/research_news/osf/sso/draft.2.html.

The DCE Security Service

A security protocol consisting of encryption keys, authentication credentials, tickets, and user passwords is used to provide secure transmission of information between two transacting parties in a DCE client/server enterprise.

by Frédéric Gittler and Anne C. Hopkins

The Open Software Foundation's Distributed Computing Environment (DCE) is a collection of integrated services that support the distribution of applications on multiple machines across a network. In most cases, networks are inherently insecure because it is possible for someone to listen to traffic or behave as an impostor. Without countermeasures this threat could prohibit the distribution of business applications.

The DCE security service described in this article provides a set of security mechanisms that can be easily used by a distributed application to remove the security vulnerabilities mentioned above.

The security functionality provided by the DCE security service includes:

- Identification and authentication of users to verify that they are who they claim to be
- Authorization for applications to decide if a user can access an operation or object
- Secure data communications to protect the data communication of an application against tampering or eavesdropping.

Security Services

The DCE security service, with additional new services and facilities, is based on the Kerberos system.[1] The Kerberos system performs authentication of users and servers based on cryptographic keys so that communicating parties can trust the identity of the other. DCE augments Kerberos with a way to transfer additional security attributes (beyond just identity) to a server which may choose to perform access control on those attributes. The DCE communication protocol contains support for protected communications that relies on crytographic session keys provided by Kerberos.

Fig. 1 shows the environment in which the DCE security service operates, and the services provided on the DCE security server.

Registry. Every DCE security service user is known as a principal. Interactive (human) users, systems (computers), and application servers (processes) are all principals. Each principal shares a secret key† with the DCE security server. The secret key for interactive users is derived from the user password. This security model relies on the fact that a particular key is known only to the principal and the DCE security service.

† As in the Kerberos system, keys are used for encrypting and decrypting data transferred in a network transaction, and are known only to the DCE security server and the parties involved in the transaction.

The registry service is the manager of the central registry database which contains the principal's name, universal unique identifier (UUID), secret key, UNIX® account attributes, and other attributes of the principals. These attributes include the extended registry attributes (ERA), which may be defined and instantiated by an administrator.

Like other DCE services, access to the registry service is based on the use of remote procedure calls (RPCs). The registry's operation is secure because it uses a protected RPC for all of its transactions. Extended registry attributes are covered in more detail later in this article.

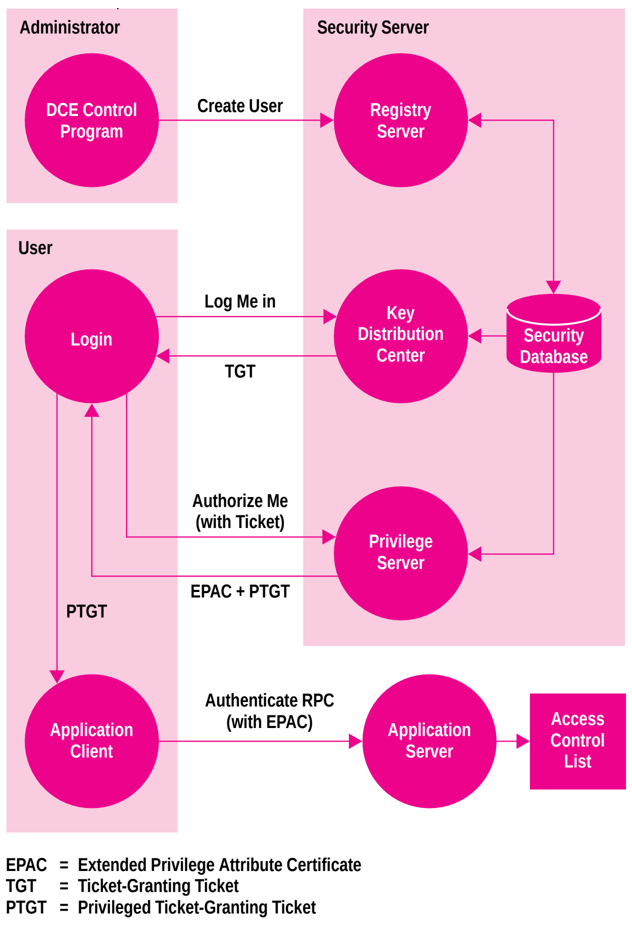


Fig. 1. The components of the DCE security server in relation to the other components typically found in a distributed environment.

Glossary

The following are some of the terminology and associated acronyms frequently used in this article.

Extended Privilege Attribute Certificate (EPAC). A credential provided by the DCE privilege service containing user and group identities and attribute-value pairs. This information is used by an application server to make authorization decisions.

Extended Registry Attribute (ERA). A mechanism in which attribute-value pairs are associated with principals. The information in these attribute-value pairs may be used to deny or grant an authorization request.

Principal. An entity such as a user, an application, or a system whose identity can be authenticated.

Service Ticket. A credential used by an application client to authenticate itself to an application server.

Ticket-Granting Ticket (TGT). A credential that indicates that a user has been authenticated and is therefore eligible to get tickets to other services.

Identification and Authentication. The first interaction between a user and the DCE security service is the login sequence when the identity of a user is authenticated by a secret key. The result of this authentication is a ticket-granting ticket (TGT) containing the user principal's credentials. The TGT indicates that the user has been authenticated. It is used, as its name implies, to obtain tickets to other services. The life span of a TGT is limited to ensure that the user represented by the credentials is the user currently using the system and that the user's credentials are up-to-date.

The user and group identity and the extended registry attributes are not part of the TGT issued by the authentication service. The privilege service supports an additional authorization by providing user and group identities and attributes in the form of an extended privilege attribute certificate (EPAC). During a login sequence, after the TGT is obtained, the run-time DCE security service makes a request to the privilege server to issue a privilege TGT. This ticket is a combination of the TGT and a seal of the EPAC.

The privilege TGT is stored in the user's environment and is used by the secure communication mechanisms to obtain a service ticket from the authentication service. The service ticket is used by the communication mechanisms to perform mutual authentication between the application client and the application server.

In each of these exchanges, secret session keys, which are known only to the DCE security service server, are generated for a particular session between the client and server. The DCE security run-time environment, RPC, and GSS (Generic Security Service)[2] API use these keys for data encryption or integrity protection generation in any network communication during a particular session. A brief description of the GSS API is given later in this article.

Authorization. DCE security provides application servers with multiple options for authorization. A server can choose to grant access to a user based on one of the following three models.

- Name-based authorization. The simplest but least scalable way of doing authorization is to compare the name of the remote principal with the names stored in an application-specific database. This method is called name-based authorization and is available when using the DCE secure communication mechanisms.
- Privilege-based authorization with access control lists. DCE servers can choose to protect their resources with access control lists (ACLs). An ACL contains entries that describe the particular permissions granted to various principals. An ACL entry may specify an individual user (principal) name, a group name that implies several principals, or "other" to indicate any principal not already matching a user or group entry. Users, groups, and others from a foreign cell may also be specified in an ACL entry.

  When a server receives a remote request, it asks the authenticated RPC run-time environment for the caller's EPAC. The EPAC contains the caller's principal and group identities, which are compared against the ACL to determine if access is granted. If the caller's principal identity matches the principal in an ACL entry, and if that ACL entry contains the required permissions, then access is granted. If there is no match on the principal, but one of the caller's groups matches a group ACL entry, then the permissions in the group entry apply.

  The DCE library includes facilities to manage ACLs and perform authorization checks based on ACLs. ACLS are described in the article on page 49.
- Other authorization. Other authorization mechanisms are made possible by the ERA facility. A server can use the value of any given attribute in a user's EPAC to decide whether it should service or deny any given request.

Secure Data Communication

DCE provides the remote procedure call (RPC) communication mechanism as one of its core services. The DCE security service is designed to support protected RPC communication.

Not all distributed applications in a DCE environment will use RPC. Most client/server applications in existence today are message-based, and changing them to use the RPC paradigm is expensive and time-consuming. It is also not practical for certain applications to use RPC. These applications nonetheless require security. For this reason the DCE security service now supports the Generic Security Service API (GSS API), which allows an application to authenticate itself to a remote party and secure data for transmission over an arbitrary communication mechanism.

Four basic levels of protection are available with either RPC or GSS API:

- No protection. The DCE security service does not mediate or participate in the connection.
- Authentication. The user of the client application is authenticated to the server.
- Data integrity. A cryptographic checksum is included with the data transmitted. The DCE security service guarantees the data received is identical to the data transmitted.
- Data privacy. The data is transmitted in an encrypted form and is therefore private to the sender and the receiver. United States export regulations limit the availability of this level of protection outside of the United States and Canada.

The higher protection levels include the protections offered by the lower levels.

Security beyond DCE

Logically, two login sequences are required: the login to the system and the login to DCE. Entries in the DCE security service registry contain all the attributes associated with a UNIX account. These entries can be used instead of the traditional /etc/passwd file or NIS† database as the source of information for the UNIX system login. The HP-UX* operating system integrates the system and DCE login sequences into an integrated login facility, which is described in the article on page 34.

The DCE security service can be used as the core security service for the enterprise because it features an extensible registry through the ERA facility. Products from HP and other manufacturers licensing DCE from the Open Software Foundation (OSF) will undoubtedly use the extended registry attribute facility either to provide other integrated login facilities or to synchronize the DCE security service registry with other user databases.

The secure data communication mechanisms described above can be used by system vendors to secure the standard network communication protocols, such as the file transfer protocol (ftp).

Security Mechanisms

The mechanisms used by the DCE security service to provide secure data communication are a combination of key distribution, data encryption, and data hash tables. The purpose of this section is to give more details about these mechanisms. Some details have been omitted for brevity and readability. More formal and complete descriptions of the algorithms can be found in the references indicated below.

Data Encryption. The DCE security service uses the Data Encryption Standard (DES)[3] algorithm to protect the data it transmits. This algorithm is used by both RPC and the GSS API to protect user data and guarantee its integrity. DES requires that the two parties exchanging information share a secret key, which is only known to the two parties. This key is 64 bits long and has 56 bits of data and 8 bits of parity.

DES encrypts plain text in blocks of 64 bits. The encryption is obtained by the iteration of a basic operation which combines permutation of bits for both key and data with exclusive-OR operations. The result of the encryption is a block of cipher text in which each bit depends on all the bits of the key and the plain text. Decryption of the cipher text involves the inverse of the same basic operation. The party receiving the cipher text and performing the decryption has a copy of the key used for encryption.

The DCE security service uses DES in cipher-block-chaining mode in which plain text blocks are exclusive-ORed with the previous cipher text block before being encrypted. The DCE security service also uses confounder data, which is a dummy block of random data placed before the application data. Confounder data is used to prevent guessing by correlation between blocks of encrypted data. The same block of plain text can result in two completely different blocks of data once encrypted with the same key because of the fact that the confounder data will be different. These two techniques render security attacks particularly difficult because each block of cipher text depends on the previous cipher block and some random data.

One-Way Hash. The DCE security service uses the message digest 5 (MD5) algorithm[4] coupled with the DES encryption algorithm to guarantee the integrity of the data being transmitted and verify the success of the decryption operations. MD5 produces a 128-bit signature (also called a message digest) that represents the data being transmitted. This message digest is obtained by processing the data in blocks of 512 bits. The algorithm is driven by a fixed table containing 64 operations. It uses four 32-bit variables and involves rotation, exclusive-OR, OR, negation, AND, and addition operations on these variables and the 16 32-bit segments contained in each block. Like all one-way hash functions, MD5 is designed to be easy to compute and difficult to break (i.e., derive plain text from a given hash). DCE uses CCITT-32 CRC,[5] a checksum algorithm, to verify data integrity in certain cases.

Keys. The DCE security service uses two types of keys: long-term principal secret keys and conversation or session keys.

- Principal keys. The DCE authentication protocol (described below) requires that the DCE security server and the principal requesting authentication share a secret key. For a machine or process principal, this key is stored in a file and is protected by the local operating system protection mechanisms. In the case of a human principal, the secret key is derived from the user's password by a one-way hash function.[1] All the principal keys are stored in the DCE registry.
- Conversation or session keys. Conversation and session keys are used to encrypt the data and checksums exchanged between the application client , the application server, and the DCE security server. The designs of the DCE and Kerberos security mechanisms avoid the need to establish a long-term secret key for each pair of communicating principals by creating short-lived session keys and communicating them securely to each principal engaging in a data exchange. In addition, session keys reduce the vulnerability of long-term principal keys because the latter are used less often and therefore are less susceptible to offline attacks.

The conversation and session keys are generated as random numbers by the DCE security service and are not reused. These keys have typical lifetimes measured in minutes. Session keys are keys communicated to principals in tickets, whereas conversation keys are established dynamically by the RPC run-time environment to protect the data transmission. Session keys are used in the establishment of communication keys.

Authentication Protocol

A simplified illustration of the authentication protocol is shown in Fig. 2. The circled numbers in this section correspond to the circled numbers in Fig. 2.

At the start of a user login sequence the computer establishes a session with the DCE security service. The user's password is transformed into a secret key ①. The client system has a file containing a machine TGT and a machine session key. Knowledge about the machine session key and

†NIS, or Network Information Services, is a product from Sun Microsystems.

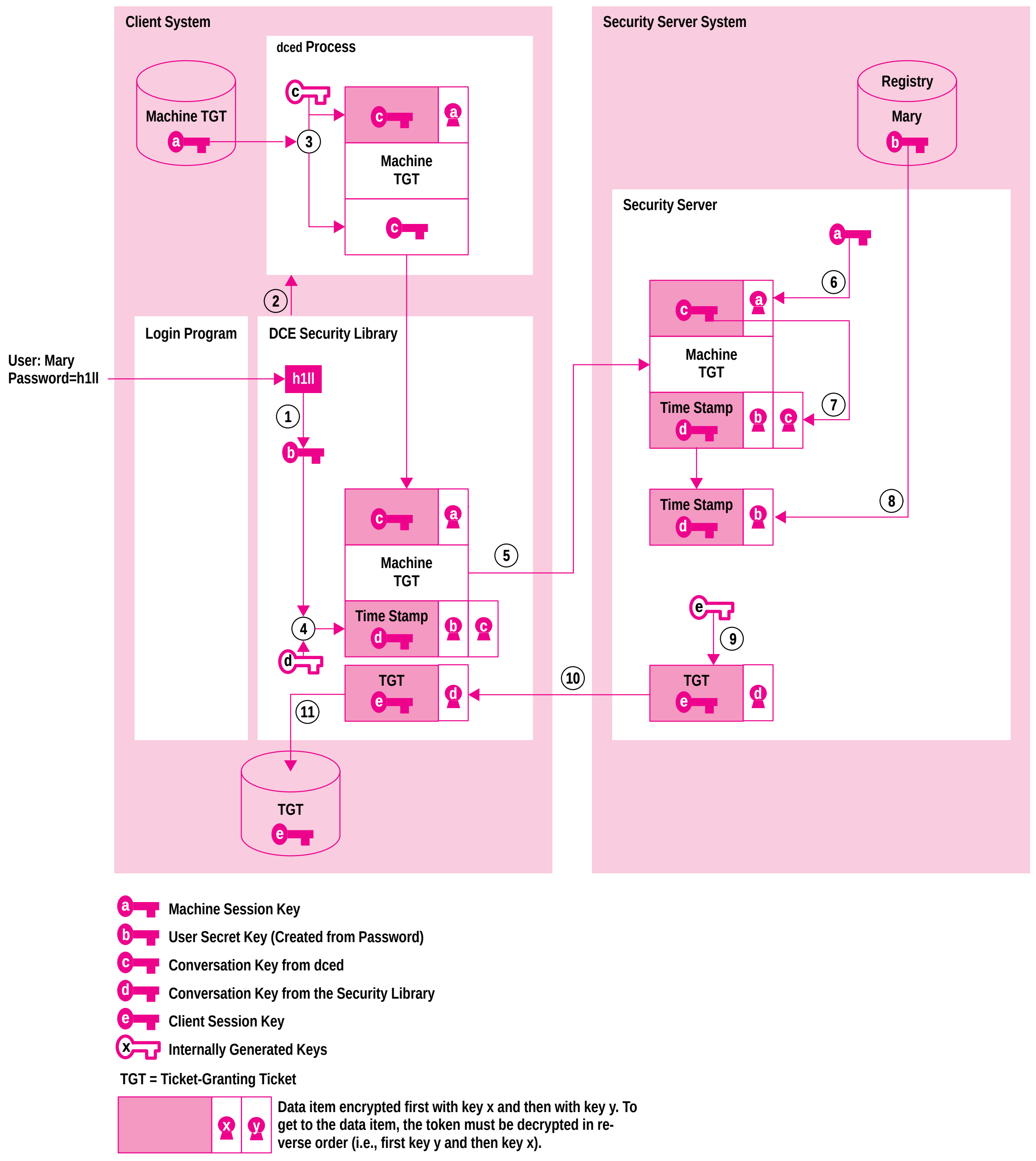

Client System
dced Process
Machine TGT
Machine TGT
Login Program
DCE Security Library
User: Mary
Password=h1ll
h1ll
Machine TGT
Time Stamp
TGT
TGT
Security Server System
Registry
Mary
Security Server
Machine TGT
Time Stamp
Time Stamp
TGT
Machine Session Key
User Secret Key (Created from Password)
Conversation Key from dced
Conversation Key from the Security Library
Client Session Key
Internally Generated Keys
TGT = Ticket-Granting Ticket
Data item encrypted first with key x and then with key y. To get to the data item, the token must be decrypted in reverse order (i.e., first key y and then key x).

Fig. 2. Creation of a ticket-granting ticket (TGT) via the authentication protocol..

the user secret key is shared between the client system and the DCE server system (see keys a and b in Fig. 2).

The protocol used for authentication is known as the DCE third-party preauthentication protocol. The protocol starts with the DCE security library requesting, on behalf of a login utility, a conversation key and a machine TGT from the DCE daemon, dced ②. Dced provides the first conversation key and the machine TGT along with a copy of the conversation key encrypted with the machine session key ③. The security library then generates a token containing a time stamp and a second conversation key. The library encrypts that token twice: once with the key derived from the user password and once with the first conversation key ④. This encrypted token is passed to the DCE security server along with the machine TGT and the encrypted conversation key received from the dced process ⑤.

Upon receipt of the token and other items, the DCE security server decrypts the first conversation key using the machine session key ⑥. It then decrypts the token containing the time stamp and the second conversation key using the first conversation key ⑦. Next, the token is decrypted using the user's secret key stored in the registry database ⑧. If the time stamp is within acceptable limits, the DCE security server creates a token containing a TGT and a client session key ⑨. The security server passes the token back to the client encrypted with the second conversation key ⑩. The client decrypts the token, validates its content, and stores the TGT and the client session key in the login context for use in future requests for service tickets ⑪.

At this point the user and the DCE security server are mutually authenticated. Note that the user's secret key was never sent (in plain or ciphered format) to the DCE security server. Proof that the user knows the correct password is verified by the fact that the time stamp is successfully decoded by the DCE security server.

Privilege Service. The TGT described above does not contain the information necessary for the advanced authorization mechanisms such as groups and ERAs. The privilege service provides this information by creating an EPAC and a privilege TGT, which contains the TGT and a seal (checksum) of the EPAC.

When an authenticated RPC is attempted and a valid privilege TGT is not available, the privilege service is contacted by the security library. First the library obtains a service ticket for the privilege service in a manner similar to what is described below, but using a TGT instead of the privilege TGT.

The privilege service then prepares the extended privilege attribute certificate, creates the privilege TGT, and communicates it back to the client. Application servers will be able to request the EPAC through the RPC run-time environment.

Secure Communication. The authentication and key exchange protocol needed to establish a secure communication channel between a client and its associated server is transparent to the application. The RPC and GSS API facilities and the DCE security service library cooperate in establishing a secure communication channel.

Fig. 3 is a simplified† representation of the sequence of events for establishing a protected RPC communication channel, assuming a valid privilege TGT has already been established by the privilege service as described above. The circled numbers in Fig. 3 correspond to the circled numbers in this section. First, the application client makes a request to the application server by calling an RPC stub①.

Since the application client needs a service ticket to authenticate itself to the application server, the security library generates a request to get a ticket and a conversation key from the security server. This results in the creation of a token containing the request for the ticket and the privilege TGT encrypted by the client session key learned during the login sequence. The token is sent to the key distribution center (KDC) which is in the security server②.

The KDC decrypts and validates the request and then generates a conversation key for use between the application client and the application server. It encrypts the conversation key and the authentication information (in the service ticket) with the secret key it shares with the application server ③. It attaches another copy of the conversation key to the service ticket and encrypts the whole structure with the client session key ④. This token is then sent to the application client system ⑤, which decrypts it and learns the conversation key ⑥.

RPC then encrypts the RPC request with the conversation key ⑦ and sends it to the application server. The application server learns the conversation key and checks the client's authenticity. To accomplish this, the application server sends a challenge, which is just a random number ⑧. The client receives this challenge and replies by sending a token containing the encrypted challenge and the encrypted service ticket and conversation key obtained from the security server ⑨. The server decrypts the ticket and obtains the client privileges and the conversation key ⑩. It decrypts the challenge with this conversation key ⑪, and if it matches what is sent, the authenticity of the client is assumed. It then proceeds to decrypt the request from the client ⑫. The client and server now share a secret conversation key.

Additional Functionality

Extended Registry Attributes

The DCE registry contains principal account data in a well-defined format (i.e., a static schema). Every account record contains the same number and types of data fields, all targeted to meet the requirements of either DCE security or UNIX platform security. To support integration with other platforms and security systems, the DCE registry needed a way to store non-DCE or non-UNIX security data for principals. To meet this need, the DCE registry was augmented with a dynamic schema facility called the extended registry attribute (ERA) facility, which supports the definition of new types of data fields called attribute types and the assignment of specific values for those attribute types to principals and other registry objects like groups and organizations.

In the ERA schema, administrators define new attribute types by specifying a unique attribute name (e.g., X.500_Distinguished_Name), the appropriate data type (e.g., string), the type of registry object (e.g., principal) that supports attributes of this type, and other related information. Once the attribute type has been defined in the schema, an administrator can attach an instance of that attribute type to any registry object that supports it. For example, an attribute instance whose type is X.500_Distinguished_Name and whose value is /C=US/o=HP/OU=OSSD/G=JOE/S=KING could be attached to the principal Joe.†† From then on applications that require knowledge of Joe's X.500 distinguished name could query the registry for that attribute type on the principal Joe.

In some cases, attribute values of a certain type are more appropriately created and maintained outside of the DCE registry. These could include attributes that are already maintained in a preexisting legacy database or attributes whose values differ depending on discriminating factors such as time of day or operation to be invoked. The ERA trigger facility supports cases such as these by providing an automatic trigger (or callout) to a remote trigger server that maintains the attributes of interest. For example, if the registry receives a

† In particular, the conversation key is established in more steps than shown, and the protocol implements caching so as not to require all steps to be executed every time.

†† See the article on page 23 for an explanation of the fields in this string.

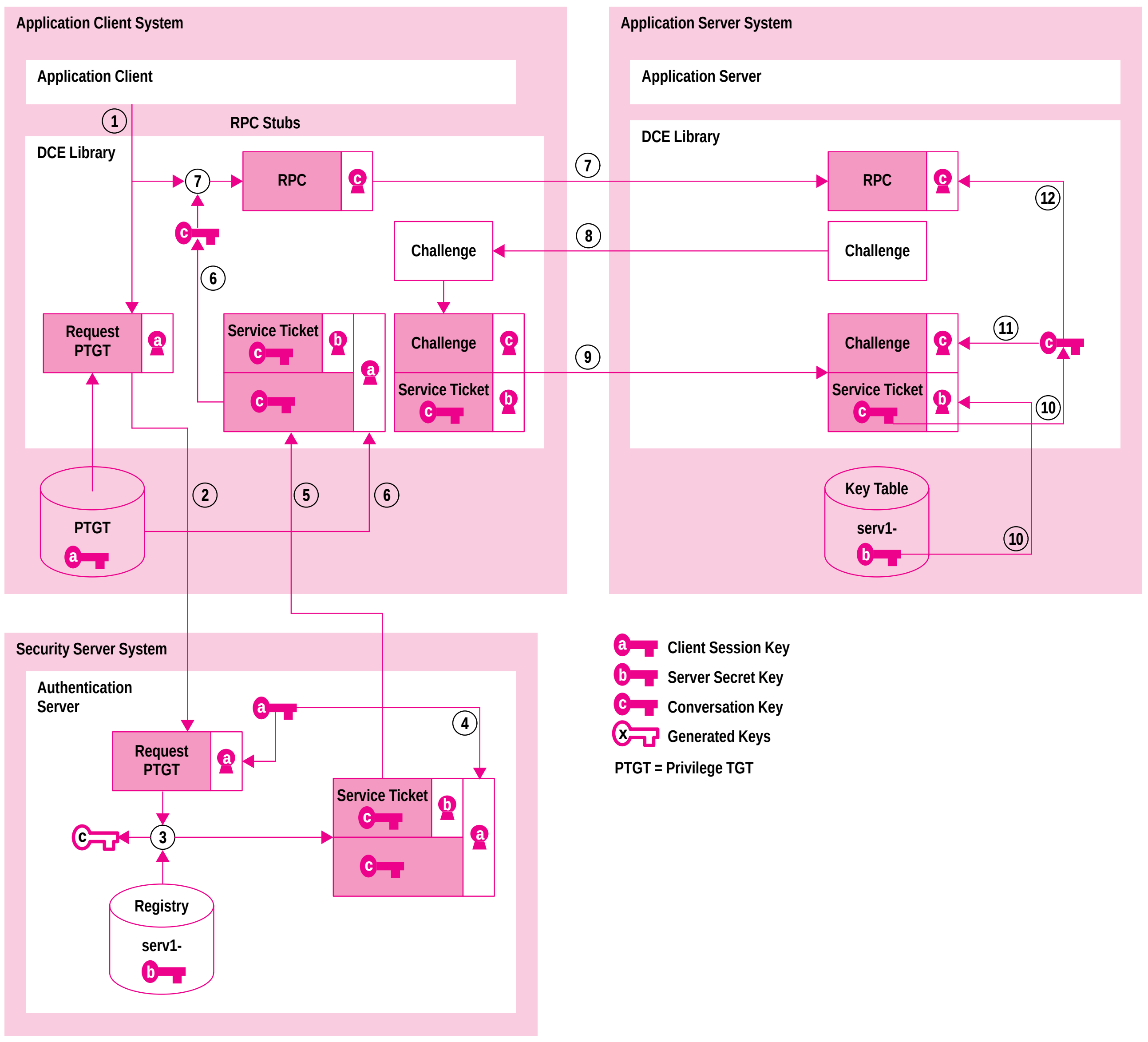


Fig. 3. Setting up a secure communication between a client and server.

query for a particular attribute type that is marked as a trigger, the registry forwards the query to a preconfigured trigger server. The server will return the appropriate attribute value to the registry, which will then respond to the original query with this value. A query for a trigger attribute may include input data required by the trigger server to determine the appropriate attribute value to return. Trigger servers are not provided as part of the DCE package; they are provided by third-party integrators of security systems. The ERA trigger facility provides the rules, interfaces, and mechanisms for integrating trigger servers with the DCE security service.

Some application servers need to make decisions, especially authorization decisions, based on the calling principal's attribute values. The DCE privilege service supports this by providing a way for applications to request that specific attributes be included in a principal's EPAC. As described earlier in the "Identification and Authentication" section, the RPC run-time environment supports queries for obtaining the calling principal's EPAC. This enables application servers to base decisions on the caller's attribute values and the identity and groupset information in the EPAC.

Delegation

In a distributed environment, an application server processing a client request may have to make a request on its own to another server to complete the client request. We will call the application server with the request an intermediate server. The identity reported by the intermediate server to the server it contacts can be either its own identity or the identity of the client that made the original request. This latter case is called delegation because the intermediate server acts as a delegate of the client. A delegation chain is built as intermediate servers call other intermediate servers.[6]

For delegation to be possible, the client has to enable this feature explicitly. Two types of delegation are available:

- Traced delegation in which the identity and privileges of each intermediary are kept and can be used for access control
- Impersonation in which only the originator's identity and privileges are carried in the extended privilege attribute certificate.

ACLs have been extended to support delegation, making it possible to grant access based not only on the originator of the request, but also on the intermediaries. This allows administrators to grant access to servers acting as delegates on behalf of particular originators without granting access to the same servers operating on their own behalf.

Compatibility with Kerberos

The authentication service provided in the DCE security is derived from Kerberos version 5.[1] The protocol used between a client and server using the DCE security service is the native Kerberos protocol and has been adapted for RPC transport.

DCE security supports Kerberos version 5 clients (e.g., a telnet, or a terminal server that uses Kerberos version 5). This removes the need to manage a separate Kerberos realm because DCE security supports the registration and authentication of Kerberos principals.

DCE security also provides an API that can be used to promote Kerberos credentials that have been forwarded to a DCE client into full DCE credentials. Full DCE credentials represent an authenticated DCE principal, thereby enabling use of DCE services.

Auditing

DCE offers an auditing service that is part of DCE security. The DCE security and time services use auditing to record security-relevant events like account creation, ticket granting, and system time changes.

DCE auditing is controlled by the DCE control program, with which DCE administrators can select the events to audit and control the operation of the audit subsystem.

Authenticated RPC

The DCE remote procedure call (RPC) facility is described in more detail in the article on page 6. The RPC facility is integrated with the DCE security service and is referred to as the authenticated RPC run-time environment.

When an application client wants to make a protected remote call, it calls the authenticated RPC run-time environment to select:

- The authentication service, which can be either no authentication or secret key authentication
- The protection level, which specifies whether authentication should occur only at the beginning of an RPC session or at each message or packet and whether message data should be integrity or confidentially protected
- The authorization service, which can be name-based, in which case only the name of the caller is known to the server, or privilege-based, in which case all the privileges of the client, in the form of an EPAC, are made available to the server for authorization.

The application developer can trade off the resources consumed by an application with the level of security required.

Generic Security Service API

The GSS API improves application portability by reducing security-mechanism-specific code. It also provides transport independence since the data protection is not tied to a particular communication mechanism (e.g., DCE RPC). GSS API calls are used to authenticate and establish a security context between communicating peers and to protect blocks of data cryptographically for transmission between them. The data protection includes data origin certification, integrity, and optionally, confidentiality.

The GSS API supports many different underlying security mechanisms. The GSS API implementation provided with DCE supports both the DCE and the Kerberos version 5 mechanisms.

Security Run-Time Environment

Applications can access security functions directly through the security library, which is part of the DCE library on the HP-UX operating system. The security library provides APIs to make access decisions based on ACLs, manage key tables, query and update registry data, login and establish credentials, and so on.

System administrators and users can use a series of commands to administer the security service or manage their local security resources such as credentials, ACLs, or key tables. Most of the administrative commands are part of the DCE control program.

Multicell Configurations

In large enterprise networks, it is often impractical or undesirable to configure a single cell. For this reason, DCE features intercell communication mechanisms. See the article on page 6 for a brief description of cells.

The DCE security service is an actor in this intercell environment. Through a mechanism of key exchange, a relationship of trust can be established between two cells. When an application client wants to communicate with a server in a foreign cell, it must obtain a service ticket for that server. To do so, the DCE security service automatically generates a foreign privilege TGT, which contains the privilege information about the principal (application client) in its local cell encrypted using the foreign cell's keys. This key, shared between the two cells, is used to authenticate and secure this protocol. The DCE security service then proceeds to get a service ticket to the foreign server by contacting the foreign authentication service as it would do for the local cell by using the foreign privilege TGT instead of the privilege TGT used in the example given earlier.

ACLs support the intercell operations by allowing foreign users, groups, and others to be granted permissions.

High Availability

The DCE security service is an essential piece of the distributed computing environment. Thus, the security service must stay operational around the clock even when systems are down or network connections are unavailable, which could happen frequently in wide area network environments.

For this purpose, the DCE security service features a server replication mechanism. The master replica is the only one that can accept requests for updates such as password changes or account modifications. These modifications are sent securely to slave replicas, which contain a duplicate image of the registry database, but support only query, not update operations. The use of slave replicas improves performance in busy environments since additional DCE security servers are available to process queries and requests for secure communication.

The DCE security service administrative commands allow the role of master to be moved between replicas. In case the machine hosting the master is not available for some time, the administrator can force a slave to become the master.

In the rare case in which no network connection is available to reach a DCE security server, the DCE security login client will use a local cache of credentials that have been granted recently to perform authentication. However, the credentials usually cannot be used to obtain service tickets.

System Security Requirements

The use of the DCE security service alone does not guarantee a secure distributed computing environment. The security service relies on protection features offered by the local operating system to store its data and credentials.

The systems hosting a DCE security server must be protected from unauthorized access. They should be placed in a secure area, such as a locked room, and be given the highest security considerations. In particular, certain network services should be disabled and a limited number of users should be given access. This security is required because the DCE security server holds the keys to all the principals in the enterprise.

The systems hosting the application servers should also be managed with care, mainly to protect the enterprise data, which is often not protected by the DCE security service.

Application clients do not need such stringent management guidelines. On multiuser systems, the user environment should be partitioned so that one user cannot steal the credentials of another active user, which could be done by reading the other user's credential files.

The DCE security service does not guarantee that there are no undetected intruders in the system. It offers no protection if the program used for login has been modified to steal the password, saving it for future retrieval by an intruder.

If a system other than one hosting a DCE security server is compromised, only the application servers residing on that system and the users who performed a login on that system during the period of compromise are affected. The overall distributed computing environment protected by the DCE security service is not affected. This is because the keys are known only by the owner (server, machine, or application) and the DCE security servers, and they are never communicated to a third party.

Acknowledgments

The DCE security service was developed by Hewlett-Packard for the Open Software Foundation as part of the Distributed Computing Environment project. The DCE security service finds its roots in the Domain operating system, the Network Computing System (NCS), and the Athena project at the Massachusetts Institute of Technology, which created Kerberos. Over the years, many different engineers and managers, too numerous to recognize by name, have worked on the product. The DCE security service continues to evolve as new services are added through the prestructured technology program under the auspices of the OSF.

The authors would like to thank Sue Kline, Mike Kong, Sean Mullan, Joe Pato, Bill Sommerfeld, and Rob Stanzel for their contributions to this article.

References

1. J. Kohl, et al., *The Kerberos Network Authentication Service*, Version 5, RFC-1510, September 1993.
2. *Generic Security Service API*, Preliminary Specification P308, X/Open Company Ltd., January 1994.
3. *Data Encryption Standard*, NBS FIPS PUB 46-1, National Bureau of Standards, U. S. Department of Commerce, January 1988.
4. R. Rivest, *The MD5 Message Digest Algorithm*, RFC-1321, April 1992.
5. *Error-Correcting Procedures for DCEs Using Asynchronous-to-Synchronous Conversions*, Recommendation V.42, CCITT, 1988.
6. M. Erdos and J. Pato, "Extending the OSF DCE Authorization System to Support Practical Delegation," *Proceedings, Privacy and Security Research Group Workshop on Network and Distributed System Security,* February 1993.

An Evolution of DCE Authorization Services

One of the strengths of the Open Software Foundation's Distributed Computing Environment is that it allows developers to consider authentication, authorization, privacy, and integrity early in the design of a client/server application. The HP implementation evolves what DCE offers to make it easier for server developers to use.

by Deborah L. Caswell

In the Open Software Foundation's Distributed Computing Environment (DCE),[1,2] services are provided by server processes. They are accessed on behalf of users by client processes often residing on a different computer. Servers need a way to ascertain whether or not the user has a right to obtain the service requested. For example, a banking service accessed through an automated teller machine has to have a way to know whether the requester is allowed to withdraw money from the account. A medical patient record service has to be able to know both who you are and what rights you should have with respect to a patient's record. A policy can be implemented such that only the patient or the legal guardian of the patient can read the record, but doctors and nurses can have read and write access to the record.

The process of determining whether or not a user has permission to perform an operation is called authorization. It is common to separate the authorization policy from the authorization mechanism. Authorization policy dictates who has permission to perform which operations on which objects. The mechanism is the general-purpose code that enforces whatever policy is specified. In DCE, the encoding of the authorization policy is stored in an access control list (ACL). Every object that is managed by a server such as a bank account or a patient record has associated with it an ACL that dictates which clients can invoke each operation defined for the object.

For example, to encode the policy that the owner of the bank account can deposit and withdraw money from the account and change the mailing address on the account, but only a bank teller may close the account, an ACL on a bank account owned by client Mary might look like:

```
user:Mary:DWM
group:teller:C
```

where `D` stands for permission to deposit, `W` for permission to withdraw, `M` for permission to change the mailing address, and `C` for permission to close the account.

Each application is free to define and name its own set of permissions. The `D`, `W`, and `C` permissions used in the example above are not used by every server. An application in which the `D` (deposit) permission makes sense could choose to name it as the "+" permission. Also, many applications will not have a deposit operation at all. Therefore, the interpretation of an ACL depends on the set of permissions defined by the server that uses it.

The first part of this paper describes the specifications and authorization mechanisms (code) offered in DCE that support the development of authorization services. The second part describes our efforts to supplement what DCE offers to make it easier for the server developer to use authorization services. The ACL functionality described here pertains to DCE releases before OSF DCE 1.1 and HP DCE 1.4.

Authorization Based on Access Control Lists

Fig. 1 shows the client/server modules required for an ACL authorization scheme used in a hypothetical bank application that was implemented using DCE. To understand the

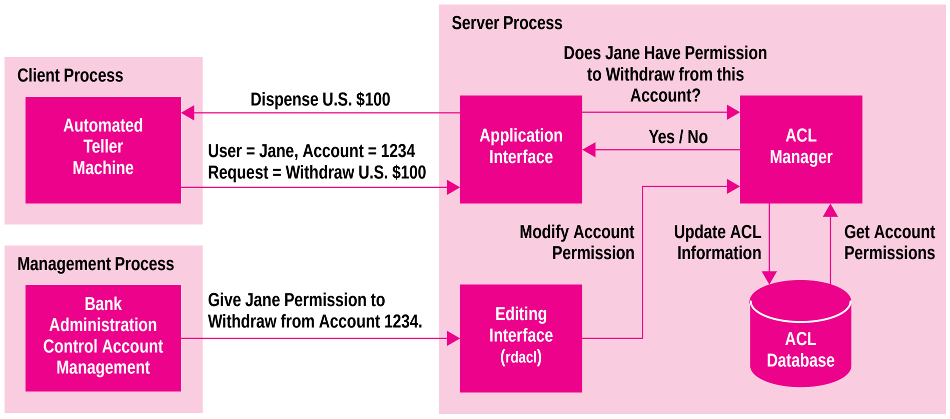

Client Process
Automated Teller Machine
Management Process
Bank Administration Control Account Management
Server Process
Dispense U.S. $100
User = Jane, Account = 1234
Request = Withdraw U.S. $100
Give Jane Permission to Withdraw from Account 1234.
Application Interface
Editing Interface (rdacl)
Does Jane Have Permission to Withdraw from this Account?
Yes / No
ACL Manager
Modify Account Permission
Update ACL Information
Get Account Permissions
ACL Database

Fig. 1. Flow of information in the bank automatic teller machine example.

interactions between these modules consider the following scenario. Jane makes a request to withdraw U.S. $100.00 from her account number 1234. The application interface passes this information to the ACL manager asking for an authorization decision. The ACL manager retrieves the authorization policy for account 1234 from the ACL database and applies the policy to derive an answer. If Jane is authorized, the machine dispenses the money.

When Jane's account is first set up, a bank employee would use an administration tool (from the management process in Fig. 1) to give Jane permission to withdraw money from account 1234. The editing interface enables the ACL manager to change the policy. The ACL manager changes a policy by retrieving the current policy, modifying it, and writing it back to the ACL database.

ACL Database. A server that needs to authorize requests must have a way to store and retrieve the ACLs that describe the access rights to the objects the server manages. One application might want to store ACLs with the objects they protect and another might want a separate ACL database. Depending on the number of objects protected and access patterns, different database implementations would be optimal. For this reason, the requirements for an ACL storage system are likely to be very dependent on the type of application.

An Authorization Decision. When an application client makes a request of the application server, control is given to the manager routine that implements the desired operation. The manager routine needs to know what set of permissions or access rights the client must possess before servicing the request.

The manager routine must supply the client's identity (Jane), the name of the protected object (Account 1234), and the desired permissions (withdraw) to a routine that executes the standard ACL authorization algorithm. If the routine returns a positive result, the server will grant the client's request (dispense U.S. $100). Note that the authorization system depends on the validity of the client's identity. Authentication is a necessary prerequisite for authorization to be meaningful.

Standard Interface for Editing. Without a standard way of administering ACLs, each server developer would have to provide an ACL administration tool, and DCE administrators would have to learn a different tool for each server that uses authorization. To avoid that problem, a standard ACL editing interface is defined so that the same tool can interact with any service that implements the standard interface.

What DCE Provides

To meet the requirements for the ACL management scheme mentioned above, DCE provides code to support ACL management for some requirements and simply defines a standard interface without providing any code for other requirements. Fig. 2 shows the main components that provide DCE ACL support within the server executable.

Unforgeable Identities. DCE provides the run-time RPC (remote procedure call) mechanism, which provides the server process with information about the client making a request. Because of the authentication services provided in DCE, the client's identity is unforgeable so that the server need not worry about an impostor.

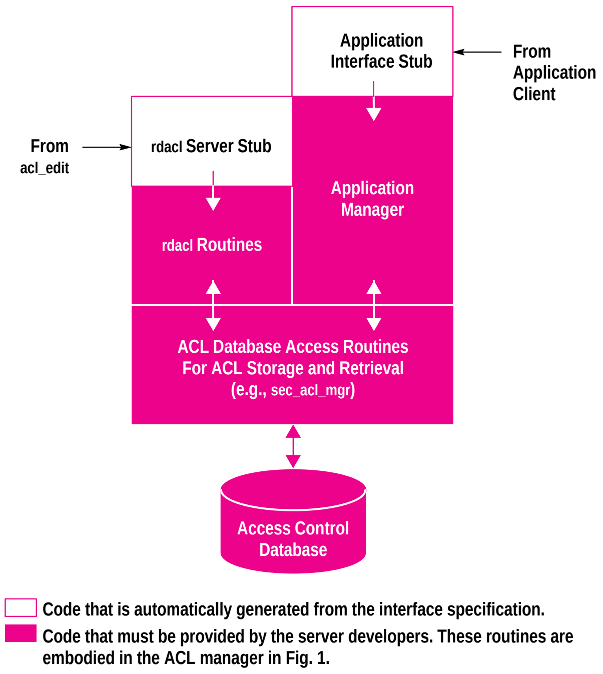

Application Interface Stub
From Application Client
From acl_edit
rdacl Server Stub
Application Manager
rdacl Routines
ACL Database Access Routines For ACL Storage and Retrieval (e.g., sec_acl_mgr)
Access Control Database
Code that is automatically generated from the interface specification.
Code that must be provided by the server developers. These routines are embodied in the ACL manager in Fig. 1.

Fig. 2. Components that provide DCE ACL support in a server executable.

ACL Database. DCE suggests an interface to an ACL storage and retrieval subsystem called sec_acl_mgr. This interface is used within the server, and therefore is not mandatory or enforceable. DCE currently does not provide an implementation of this interface for use by application developers. Furthermore, it does not contain operations for adding and deleting ACLs, so even if the sec_acl_mgr interface is used, it would have to be supplemented by other ACL database access operations.

Authorization Decisions. DCE specifies a standard way of reaching an authorization decision given a client's identity, desired operation, and authorization policy encoding. The OSF DCE 1.0 distribution for application developers does not supply an implementation of this algorithm, requiring the server developer to write the authorization algorithm.

Standard Editing Interface. DCE provides a tool called acl_edit that an administrator can use to change the authorization policy used by any server that implements the standard rdacl interface even though each server might use a different set of permissions.

DCE defines the standard rdacl interface responsible for enabling modification of the authorization policy. The rdacl interface is used by acl_edit to access and modify ACL information. DCE does not provide an implementation of the rdacl interface. Without additional help from other sources, each server developer has to write rdacl routines that call the ACL database access routines. Servers that implement the rdacl interface can be administered by any client that uses the standard interface including the acl_edit tool mentioned above.

The rdacl interface does not support adding and deleting ACLs; it only addresses editing existing ACLs. For that reason, an ACL storage subsystem must be designed and implemented for an application that supports adding, modifying, retrieving, and deleting ACLs.

The rdacl operations listed below are described in the DCE reference manual.[3] They are listed here to give an idea of the size and functionality of the interface.

- rdacl_get_access: lists the permissions granted to a principal to operate on a particular object
- rdacl_get_manager_types: gets the list of databases in which the ACL resides
- rdacl_get_printstring: gets the user description for each permission
- rdacl_get_referral: gets a reference to the primary update site
- rdacl_lookup: gets the ACL for an object
- rdacl_replace: replaces the ACL for an object
- rdacl_test_access: returns true if the principal is authorized to perform the specified operation on an object
- rdacl_test_access_on_behalf: returns true if both the caller and a specified third-party principal are authorized to perform the specified operation on an object.

An implementation of these operations has to call the retrieve and modify operations of the ACL storage subsystem, invoke the authorization decision routine, and describe the permissions that are used in the ACLs for the particular implementation.

Component Relationships. Some of the boxes in Fig. 2 represent code that is automatically generated from the interface description, and other boxes represent code that must be supplied by server developers.

The modules on the right side of the block diagram in Fig. 2 represent the application-specific interfaces and code. The application interface stub is the code generated by the Interface Definition Language (IDL) compiler when given the application interface files. For example, if we have a bank account server, the application interface stubs would receive the call and direct it to the application manager. The application managers are the modules that implement the application server functionality. In our bank example, this is the code that implements the deposit and withdrawal operations.

On the left side of Fig. 2 is the code that is specific to ACL management of the DCE standard rdacl interface. The rdaclif (rdacl interface file) server stubs are generated by running the IDL compiler over the rdaclif.idl file which is delivered with the DCE product. The rdacl routines implement the operations defined in the rdacl interface. The bottom of Fig. 2 shows the ACL storage and retrieval code. The rdacl routines make calls to the storage layer either to get the ACLs that will be sent over the wire to a requesting client or to replace a new ACL received from an ACL administration tool. The database access routines must also implement the standard ACL checking authorization algorithm and a routine to compute the effective permissions of a client with respect to a specific object. The application managers call the database access layer to get an authorization decision. For example, the code that implements the withdrawal operation needs to first make sure that the client making the request is authorized to withdraw money from a particular account.

Although they do not interact directly with each other, the application manager routines and rdacl routines coexist within the same process and call common ACL manager routines.

Summary

DCE supports a server process's ability to make an authorization decision in several ways, but as shown in Fig. 2, there is a lot of code left for the server developer to write. Some of the required code, such as the authorization decision routine, can be reused in other applications because it is application independent. Other code, such as the storage subsystem, is more application-specific and might have to be developed for each new service.

Help for the Server Developer

This section describes three evolutionary steps that we took to supplement DCE's authorization support. The approach we took to each step is not novel. Each approach has value by itself in addition to being a stepping stone to a more sophisticated approach.

Note that although the outputs from each of these steps did not directly become products, they did form the basis for HP Object-Oriented DCE (HP OODCE). HP OODCE is briefly described later in this article and completely described in the article on page 55.

Sample Applications. The first step was simply to provide an example of server code that performs ACL management. The application acl_manager is one of a set of sample applications written to demonstrate the use of various DCE facilities. These sample applications are a valuable learning tool and are also useful for cutting and pasting working code into a real-world application.

The acl_manager is based on the ACL manager reference implementation distributed with DCE source code. The sample application uses a static table of ACLs, and there is no operation for adding or deleting ACLs and no general storage manager. However, acl_edit can interact with this primitive ACL manager to view or modify the ACL for one of these static objects .

The acl_manager includes a description of how to tailor the code to one's own application server and provides more background on how ACL management works than is available in the DCE manual set.

Another sample application, the phone database, demonstrates the use of an ACL manager inside an application. This more complex sample application demonstrates how application interfaces and the ACL management interface coexist within the same server and how they interact. The phone database application uses an in-memory binary tree storage facility with a simple checkpoint facility for committing changes to stable storage. The persistent representation of ACLs can be modified by an editor for bulk input. At startup, the server parses and interprets this file.

As mentioned before, in addition to being a valuable learning tool, the sample applications provide reuse of code and ideas at the source-code level.

Common ACL Management Module Interface. Cutting a sample application and pasting it into a new application with an understanding of how it needs to be modified is surely better than starting from scratch. Reuse through a code library is better yet. The problem was how to provide a single library for ACL management when so much of it is application-specific. There is so much flexibility in how ACLs are managed. We wondered if it were possible to anticipate what most developers would need and if we would be able to satisfy those needs by creating a general-purpose library.

The first task was to partition the aspects of ACL management into those that are application-specific and those that are application independent. The application independent portion would be provided as library routines. Our approach to the application-specific portions was threefold:

- Limit the flexibility by providing routines that would be sufficient for most developers. For example, although DCE allows a server to implement more than 32 permissions, limiting support to 32 or less simplified the design considerably.
- Parameterize routines such that their behavior can be determined when the library is initialized at startup. For example, each application defines its own set of permissions. A table of permissions can be downloaded into the library rather than hard-coded into the library routines.
- Identify a well-defined interface to the storage and retrieval routines. As mentioned earlier, the storage requirements are the one aspect of ACL management that will vary the most among applications. By partitioning the functionality in this way, customers with special storage needs can write their own ACL storage management, and provided that they conform to the published interface guidelines, would still be able to use the library for other ACL management functions.

Fig. 3 shows a different view of the ACL components depicted in Fig. 2. The application server component is not called out separately in Fig. 2. The server initialization code (server.c) is typically located in this component. The application server also contains the code that directs the DCE runtime code to start listening for incoming client requests.

The application manager component in Fig. 3 contains the same functionality as the application manager component shown in Fig. 2.

The ACL manager component in Fig. 3 represents the code needed to support the rdacl interface, the ACL checking algorithm, the computing of effective permissions, and other general utilities. Basically, the ACL manager contains all the ACL code that is independent of how an ACL is stored within a database. It also encapsulates the implementation of the ACL structure itself. In other words, if the data structure that represents an ACL were to change, only the ACL manager component would need to be rewritten to accommodate the changes.

The ACL storage manager contains the ACL database access routines and the ACL database. The ACL storage manager can manage ACL storage in memory, on disk, or a hybrid of the two.

The circled numbers in Fig. 3 correspond to the following interactions between ACL manager components.

1. The application server must call the ACL manager to initialize its internal data structures and to download application-specific information such as permission print strings and reference monitor callback functions. The reference monitor implements a general security policy that screens incoming requests based on the client's identity and the authentication or authorization policies it is using. The monitor does not base an authorization decision on the requested operation or the target object. The ACL manager performs that job. A default reference monitor is provided by the ACL manager. If an application has its own reference monitor, it will be invoked instead of the default monitor supplied with the ACL manager.

2. The application server must call the ACL storage manager to allow it to initialize itself. The initialization calls performed by the application server are only done once when the whole system is initialized.

3. The application manager calls the ACL manager to perform an authorization decision or to invoke a general ACL utility.

4. The application manager calls the ACL storage manager to add a new ACL to the database or to delete an old ACL from the database.

5. The ACL manager calls the ACL storage manager to transfer an ACL to or from the database in response to rdacl requests coming from acl_edit.

6. The ACL storage manager calls the ACL manager utility routines to manipulate ACL data structures. One manipulation operation involves converting permissions from human readable form into a bitmap and vice versa.

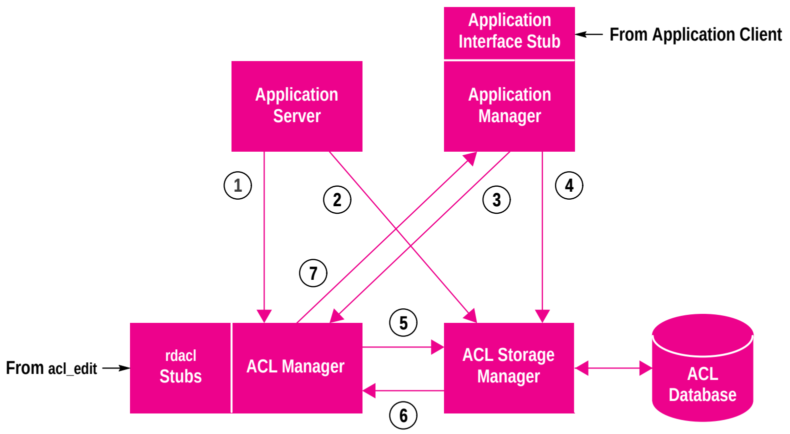


Fig. 3. Architecture for modules that make up the common ACL management module interface.

7. The ACL manager must make a callback to an application-specific reference monitor routine to screen an incoming rdacl request according to the application's general security policy.

The goal for the common ACL management module interface was to explore appropriate programmatic interfaces. Our implementation was a proof of the concept for the design and was not intended to be the best ACL manager package. The implementation provided the same functionality as the sample application except that it used an in-memory binary tree to allow applications to add ACLs at run time. The main contribution of the common ACL management module interface from an application developer's standpoint is the ability to link with a general-purpose library rather than cutting and pasting source code. The application developer can use higher-level interfaces for creating ACLs and get authorization decisions without having to understand and write the underlying mechanism.

Although the common ACL management module interface was never sold as an HP product, it was useful in several ways. First, we learned a great deal about ACL management and what developers would want to be able to do with it. Second, we used the modules in an internal DCE training class that allowed us to teach ACL management concepts and have the students add ACL management to an application they developed during a two-to-three-hour laboratory exercise. The common ACL management module interface allowed the students to spend their lab time reinforcing the concepts presented in the lecture rather than getting bogged down in writing a lot of supporting code just to make their application work. The experiences of the class reinforced our belief that it is possible to support application developers in the creation of ACL management functionality without every developer having to understand all of the complicated details of ACL management that are DCE-prescribed but not application-specific.

The version of DCE provided by OSF only supports C programmatic interfaces. It made sense to implement the common ACL management modules in C for two reasons:

- Since we were layering on top of DCE, it was more convenient to use the supported language.
- We expected that users of the common ACL management modules would also be programming in C, and so would want C interfaces to the common ACL management module interface library.

However, there is growing interest in C++ interfaces to DCE as well as support for object-oriented programming. In response to that need, a C++ class library for DCE called OODCE (object-oriented DCE) has been developed.

HP OODCE: A C++ Class Library for DCE

The common management module interface acted as a springboard for design and implementation of the C++ ACL management classes which are part of HP's OODCE product. Since it is much easier to create abstract interface definitions in C++ than in C, these DCE ACL management classes make it easier to provide access control within a DCE server. Application developers can reimplement specific classes to customize the ACL manager to fit their needs. The classes supplied with OODCE and their interrelationships are shown in Fig. 4. The classes shown in Fig. 4 represent further modularization of the ACL manager and ACL storage manager components shown in Fig. 3.

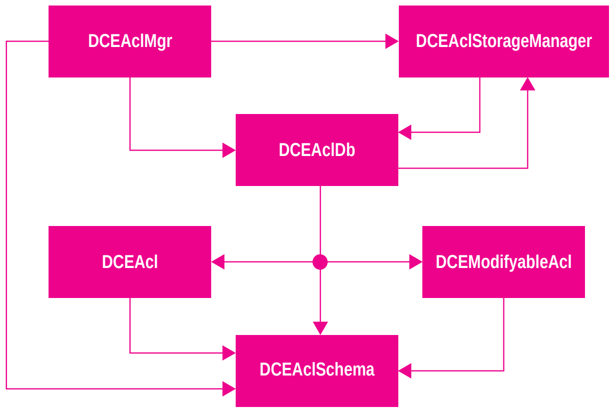


Fig. 4. HP OODCE ACL management class interrelationships.

Class Descriptions. The DCEAclMgr class implements the rdacl interface for use by the acl_edit tool and other management tools. There is one instance of an DCEAclMgr per application server. The DCEAclStorageManager manages all ACL databases for this server. The DCEAclStorageManager is responsible for finding the database in which the ACL is stored and returning a handle to that database. Programs invoke the DCEAclStorageManager interface to create or register a new ACL database and to access existing ones.

The DCEAclDB class defines the interface to an ACL database. An ACL database may define multiple 32-bit words of permissions. The interpretation of the permission bits is stored in a DCEAclSchema object, and each database has exactly one DCEAclSchema associated with it.

The DCEAcl class defines an interface for accessing DCE ACL information. In addition to the DCE ACL information, the DCEAcl class contains information about the database in which it resides, the owner and group of the protected object, and other information that is needed by an implementation. The DCEAcl's state is read-only. The DCEModifyableAcl class is a modifiable version of the DCEAcl class.

Using the OODCE ACL Management Classes. The application server invokes a simple macro that initializes the ACL system. OODCE, by default, handles all the details of making the rdacl interface ready to be invoked by remote clients. This includes registering the interface with the RPC run-time routines so that an incoming request for that interface is received and ensuring that the correct entry point for the rdacl routines is invoked. The application server also handles exporting location information to the endpoint mapper† and CDS (cell directory service) database so that clients can find the server's ACL management interface. That is the only required involvement of the application server. However, the application server may create DCEAclDb objects that can be shared across manager objects. These databases must be registered with the DCEAclStorageManager.

† The mapper maintains a list of interfaces and the corresponding port numbers where services of the interfaces are listening.

Application managers create new ACL objects by first requesting the DCEAclDb object to create a DCEModifyableAcl object and adding ACL entries to it. When done, the DCEModifyableAcl object is committed (added) to the database. To get an authorization decision, an application manager retrieves an ACL object from the database and interacts with it to get an authorization decision.

Overall, it is easier for the application developer to use the OODCE ACL manager classes than any of the previous solutions. Many of the routine tasks are done by default by the library, but they can be overridden if there are special circumstances. The ACL management objects are written to the abstract class definition so that users can provide their own implementations of DCEAclDb, DCEAcl, and DCEModifyableAcl classes and have them plug into the rest of the system.

A DCEAclDb implementation encapsulates the database access. This allows the flexibility of storing ACLs either with the objects managed by the server or in some other database. Any commercial database product can be used. The server developer need only implement DCEAclDb so that it conforms to the abstract interface and makes the calls to the commercial database of choice.

The DCEModifyableAcl class allows for fine-grained editing. The rdacl interface only supports the atomic replacement of an entire ACL, whereas the DCEModifyableAcl design supports changing individual elements within an ACL.

HP OODCE ACL objects are more general-purpose than the common ACL management module interface described earlier because the abstract class design of HP OODCE accommodates more features. Its design supports more than 32 permissions, and registration of the rdacl interface with CDS and the endpoint mapper is automatic and transparent to the server developer.

Current Status. HP OODCE is now a product. It includes default implementations for all the classes, but we expect that customers will write their own implementations of DCEAclDb and possibly of DCEAcl and DCEModifyableAcl. There is still much to learn about what distributed application developers really need from an ACL management package, but with the HP OODCE library as a product, we have more opportunity to get feedback. HP OODCE is described in more detail in the article on page 55.

Acknowledgments
I would like to thank Bob Fraley who is the codeveloper, my object-orientation mentor for the HP OODCE portion of this project, and the principal reviewer of this paper. Jeff Morgan and John Dilley were developers of HP OODCE and contributed to discussions of design and implementation of the ACL management portion. Mickey Gittler enhanced and made the HP OODCE ACL manager classes into a product. Thanks also to Jeff Morgan and Cas Caswell who reviewed this paper and gave helpful suggestions for improvement.

References

1. W. Rosenberry, D. Kenney, and G. Fisher, *Understanding DCE*, O'Reilly & Associates, Inc., September 1992.
2. J. Shirley, *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., June 1992.
3. *DCE Application Development Reference Manual*, Open Software Foundation, Cambridge, Massachusetts, 1991.

An Object-Oriented Application Framework for DCE-Based Systems

Using the Interface Definition Language compiler and the C++ class library, the HP OODCE product provides objects and abstractions that support the DCE model and facilitate the development of object-oriented distributed applications.

by Mihaela C. Gittler, Michael Z. Luo, and Luis M. Maldonado

HP's Object-Oriented DCE (HP OODCE) provides a library of framework and utility C++ classes that hide DCE programmatic complexity from developers and provide automatic default behavior to ease the development of distributed applications. The default behavior is also a great help in shortening application development time. HP OODCE offers flexibility by allowing developers to use subclassing and customized implementation. Fig. 1 shows the product structure for HP OODCE.

HP OODCE allows clients to view remote objects as C++ objects and to access member functions and receive results without making explicit remote procedure calls (RPCs). Also, applications can communicate with each other using interfaces specified by the Interface Definition Language (IDL). Finally, HP OODCE uses the C++ class library and the IDL compiler (idl++) to create an object-oriented programming environment that supports RPC-based communications, client/server classes, POSIX threads, and access to the DCE naming and security services.

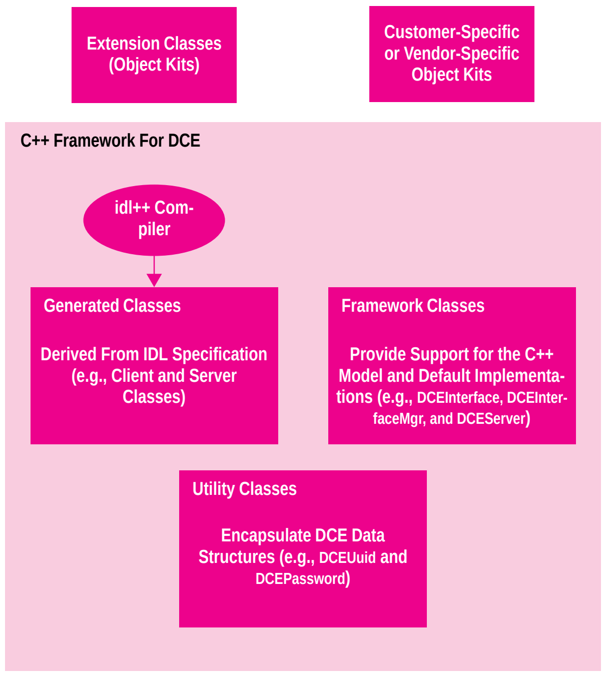

Extension Classes (Object Kits)
Customer-Specific or Vendor-Specific Object Kits
C++ Framework For DCE
idl++ Compiler
Generated Classes
Derived From IDL Specification (e.g., Client and Server Classes)
Framework Classes
Provide Support for the C++ Model and Default Implementations (e.g., DCEInterface, DCEInterfaceMgr, and DCEServer)
Utility Classes
Encapsulate DCE Data Structures (e.g., DCEUuid and DCEPassword)

Fig. 1. HP OODCE product structure.

idl++-Generated Classes

The idl++ compiler takes an IDL specification like the one shown in Fig. 2 and generates the C++ classes shown in Fig. 3. The idl++ compiler also generates the header file and stubs normally produced by the DCE IDL compiler.

The concrete client class* describes the client proxy object that accesses remote C++ objects implemented by the server. The proxy object gives the client the impression that the instantiation of a particular server object is executing locally. Fig. 4 shows an example of a client proxy class declaration for an interface to the Sleep function, which is responsible for putting a process to sleep. This class contains multiple constructors that, when called, locate the compatible manager (server) objects based on location information and the UUID (universal unique identifier) supplied as arguments to the constructors.

The abstract server class in Fig. 3 provides declarations for member functions defined in the IDL specification that correspond to remote operations that can be accessed by the client proxy object. The default concrete server class declares the member functions specified in the abstract class. The functions must be implemented by the application developer. Fig. 5 shows the abstract and concrete server manager declarations for the Sleep function.

The entry point vector contains entry points for each remote procedure defined in the IDL specification.

HP OODCE Server and Client Classes

The server code that interacts with the DCE subsystems is embodied in the DCEServer class. An instance of the DCEServer class, called theServer, manages the remote objects that are exported by the DCE server application. These objects are

```
//foo.idl
[uuid(DOFCDD70-7DCB-11CB-BDDD-08000920E4CC),
 version(1.0)]
interface sleeper
{
[idempotent] void Sleep
   (                              [in] handle_t h
                                  [in] long      time),
}
```

Fig. 2. IDL specification for the interface Sleep.

* See glossary on page 60 for a brief description of the C++ terminology used in this article.

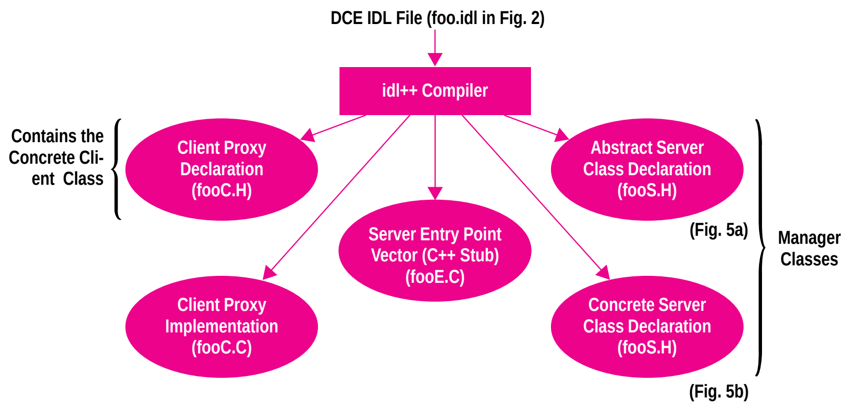

DCE IDL File (foo.idl in Fig. 2)
idl++ Compiler
Contains the Concrete Client Class
Client Proxy Declaration (fooC.H)
Abstract Server Class Declaration (fooS.H)
Server Entry Point Vector (C++ Stub) (fooE.C)
(Fig. 5a)
Manager Classes
Client Proxy Implementation (fooC.C)
Concrete Server Class Declaration (fooS.H)
(Fig. 5b)

Fig. 3. The files created after an IDL specification is processed by the idl++ compiler.

instances of the concrete server manager classes and each has a DCE UUID. There is one DCEServer instance per DCE rpc_server_listen call (currently per UNIX® process), which starts the server's run-time listening for incoming RPC requests. DCEServer has member functions that establish policies such as object registration with the RPC run-time process or the naming service and setting security preferences. Object registration takes place whenever the DCEServer class method RegisterObject is called. Fig. 6 shows the server main program for the Sleep object and the implementation of the Sleep function.

In HP OODCE, server objects are accessed via a client object (see Fig. 7). The client RPC request specifies a binding handle that locates the interface and the DCE object UUID. The entry point vector code locates the correct instance of the requested manager object. Fig. 8 shows the HP OODCE client/server run-time organization.

The idl++-generated client proxy class has methods corresponding to the operations defined in the IDL specification. Idl++ provides an implementation of the client proxy object methods. These methods locate the server and call the corresponding C ++ stub generated by the idl++ compiler. The proxy implementation handles rebinding, sets security preferences, and maps DCE exceptions returned by RPC into C++ exceptions (described below).

```
class sleeper_1_0 : public DCEInterface {
 public:
   sleeper_1_0(DCEUuid& to = NullUuid):
     DCEInterface(sleeper_v1_0_c_ifspec, to) { }
   sleeper_1_0(rpc_binding_handle_t bh, DCEUuid& to = NullUuid) :
     DCEInterface(sleeper_v1_0_c_ifspec, bh, to) { }
   sleeper_1_0(rpc_binding_vector_t* bvec) :
     DCEInterface(sleeper_v1_0_c_ifspec, bvec) { }
   sleeper_1_0(unsigned char* name,
                         unsigned32 syntax = rpc_c_ns_syntax_default,
                         DCEUuid& to = NullUuid) :
     DCEInterface(sleeper_v1_0_c_ifspec, na,e, syntax, to) { }
   sleeper_1_0(unsigned char* netaddr,
                         unsigned char* protseq, DCEUuid& to = NullUuid) :
     DCEInterface(sleeper_v1_0_c_ifspec, netaddr, protseq, to) { }
   sleeper_1_0(DCEObjRefT* ref) :
     DCEInterface(sleeper_v1_0_c_ifspec, ref) { }

   // Member functions for client
   void Sleep(
     /* [in] */ idl_long_int time
     ) ;
};
```

Fig. 4. Client proxy class declaration. The class contains several constructors for the Sleep function. The highlighted constructor is the one used in the examples in this article.

HP OODCE Framework and Utility Classes

The framework classes represent the HP OODCE object model abstraction and provide the basis for DCE functionality and default behavior (see Fig. 9). Classes, such as DCEServer, DCEInterfaceMgr, and DCEInterface interact with DCE through the DCE application programming interface.

The idl++-generated manager classes (server side) inherit from the DCEObj and DCEInterfaceMgr classes. DCEObj associates a C++ object instance, which may export several DCE interfaces, with a specific DCE object. Each DCE object is identified by its object UUID. DCEObj holds the UUID for the DCE object (see Fig. 5b).

```
      class sleeper_1_0_ABS : public virtual DCEObj, DCEInterfaceMgr {
        public:
          // Class constructors must initialize virtual base classes
          sleeper_1_0_ABS(uuid_t* obj, uuid* type) :
 ①              DCEObj(obj),
                DCEInterfaceMgr(sleeper_v1_0_s_ifspec, (DCEObj&)*this, type,
                                (rpc_mgr_epv_t)(&sleeper_v1_0_mgr)) { }

          sleeper_1_0_ABS(uuid_t* type) :
 ②              DCEObj((uuid_t*)(0)),
                DCEInterfaceMgr(sleeper_v1_0_s_ifspec, (DCEObj&)*this, type,
                                (rpc_mgr_epv_t)(&sleeper_v1_0_mgr)) { }

          // Pure virtual member functions corresponding to remote procedures
          virtual void Sleep(
                /* [in] */ idl_longint time
                ) = 0 ;
      };
(a)

      class sleeper_1_0_Mgr : public sleeper_1_0_ABS {
        public:
          // Class constructors pass constructor arguments to base classes
          sleeper_1_0_Mgr(uuid_t* obj) :
 ③              DCEObj(obj),
                sleeper_1_0_ABS(obj, (uuid_t*)(0)) { }

          sleeper_1_0_Mgr( ) :
 ④              DCEObj((uuid_t*)(0)),
                sleeper_1_0_ABS(uuid_t*)(0)) { }

          virtual void Sleep(// This is what the developer must implement
                /* [in] */ idl_long_int time
                ) ;
      } ;
(b)

 ③ Corresponds to ①
 ④ Corresponds to ②
```

Fig. 5. File fooS.H server-side declarations generated by idl++.(a) An example of an abstract server manager declaration. (b) An example of a concrete server manager declaration.

```
void main( )
{
    try  {          // Handle exceptions from constructor or DCE calls
(1)      sleeper_1_0_Mgr * sleeper = new sleeper_1_0_Mgr ; // Dynamic UUID
         DCEPthread * exitThd = new DCEPthread(DCEServer : : ServerCleanup, 0) ;

         // theServer–>SetName(”/ . ; /mysleeper”) ;
(2)      // Register Sleeper object with server object
         theServer–>RegisterObject(sleeper) ;

         // Accept all other defaults and activate the server
(3)      // Defaults are : Use all protocols, don’t use CDS, no security
         theServer–>Listen( ) ;
    }
    // Catch any DCE related errors and print out on message if any occur
    catch (DCEErr& exc) {
         traceobj < < ”Caught DCE DCEException \ n” < < (const char*)exec ;
    }
    // Destructors are called at this point and take care of DCE cleanup
}
```

(a)

```
// Developer simply implements one method to provide the implementation

void sleeper_v1_0_Mgr : : Sleep(long int time) {
    // Call the (reentrant!) libc sleep function
    sleep(time) ;
}
```

(b)

(1) Instance of Concrete Server Class

(2) Register Interface with the Object

(3) Setup for Listen

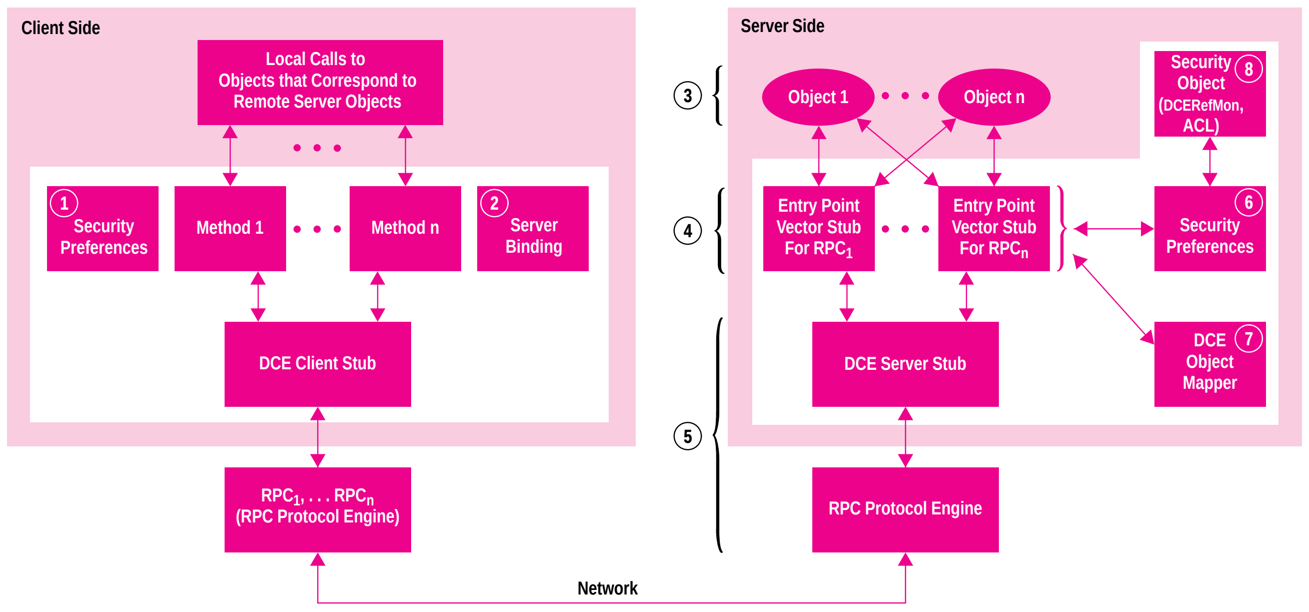
Fig. 6. (a) The server program that handles requests for the Sleep interface. (b) The implementation of the Sleep function.

```
                                                      Instance of a Client Proxy
                                                      Concrete Class
main(int, char** argv)
{
    try  {          // Handle exceptions from constructor or DCE calls

       // Constructor takes a network address and protocol sequence
       sleeper_1_0sleepClient( (unsigned  char*)argv [1]
                              (unsigned  char*)“ip”) ;

       // The Sleep method invokes the remote procedure on the server
       sleepClient.Sleep(10) ;
    }

    catch (DCEErr& exec) {
       printf(“DCEException: %s\ n”, (const char*)exec) ;
    }
    exit(0) ;
};
```

Fig. 7. A client main program that invokes the Sleep function on the server.

DCEInterfaceMgr is an abstract base class used by the server side of the application to encapsulate object and type information as well as the entry point vector called by the RPC subsystem when an incoming RPC is received (see Fig. 5a). The manager interface is registered with the DCE run-time setup and optionally with the naming service. DCEInterfaceMgr can retrieve the UUID of a particular implementation object instance, the entry point vector, and the pointer to the security reference monitor described by the DCERefMon class.

DCEInterface is an abstract base class used by the client side of the application. This class controls binding and security policies and can retrieve object references. The idl++-generated



Client Side
Local Calls to Objects that Correspond to Remote Server Objects
1 Security Preferences
Method 1
Method n
2 Server Binding
DCE Client Stub
RPC$_1$, . . . RPC$_n$ (RPC Protocol Engine)
Network
Server Side
3
Object 1
Object n
Security Object (DCERefMon, ACL) 8
4
Entry Point Vector Stub For RPC$_1$
Entry Point Vector Stub For RPC$_n$
6 Security Preferences
5
DCE Server Stub
DCE Object Mapper 7
RPC Protocol Engine

(1) Sets up security preferences, which have to be compatible with the server’s security preferences

(2) Obtains the binding handle to the server

(3) C++ object instances defined in the IDL Interface

(4) C++ entry point vectors generated by idl++

(5) RPC run-time server endpoint and server stub

(6) Checks security preferences before allowing the request access to the selected object

(7) Locates objects

(8) Set up by user

Fig. 8. The HP OODCE client/server architecture.

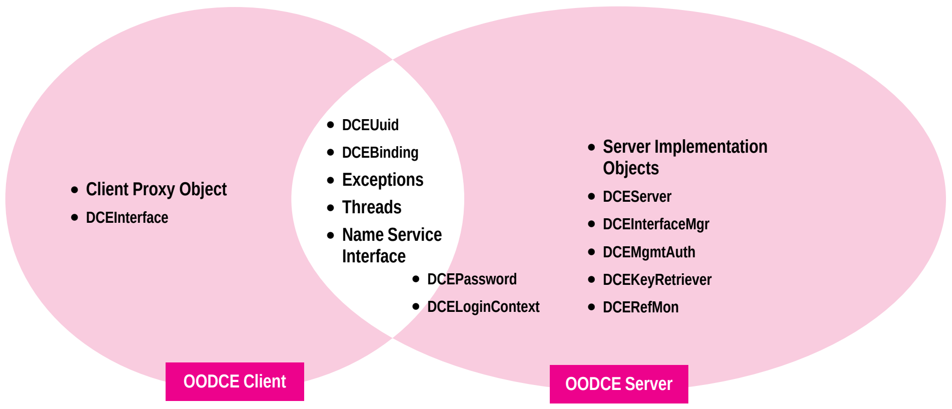


Fig. 9. HP OODCE framework and utility class library components.

client proxy class inherits from the DCEInterface class (see Fig. 4).

The HP OODCE utility classes add convenience to the HP OODCE development environment. These classes encapsulate DCE types and provide direct DCE functionality. For example, DCEUuid deals with the DCE C language representation of the uuid_t type* and its possible conversions to other types, while DCEBinding encapsulates DCE binding handle types.

Other utility classes include:

- Security services: DCERefMon for setting security preferences and DCERegistry for accessing the DCE registry database
- Naming services to model and access objects in the directory namespace
- Thread services to encapsulate the use of pthread mutexes,** condition variables, and thread policies
- Error handling and tracing services to support an exception mechanism and log information.

The security, naming, and thread services are described in the articles on pages, 41, 28, and 6 respectively.

Additional Classes

Additional classes can be derived from the abstract manager class to allow for multiple implementations for a given DCE interface. Each class must be registered with the global server (DCEServer) via the theServer object (remember that theServer is an instance of the DCEServer class). This allows the entry point vector code to locate the object manager instance, verify security preferences, and allow access to the manager methods (see Fig. 8). If the manager object is not immediately located in the HP OODCE internal map managed by theServer object, the entry point vector code can call a user-defined method to activate the manager object according to user-defined polices. Once activated, the manager object is reregistered with theServer and mapped into the object map. An object manager can be deactivated (removed from the object map) when requested by the user application.

While HP OODCE adheres to the object model provided by DCE, two extensions have been made to enhance object functionality. An ObjRef class contains a reference to an object and may be used to pass remote object identities between remote objects. When an ObjRef is used to establish the binding to an object, the referenced object may need to be activated by bringing its persistent data into memory from a file. HP OODCE provides an activation structure that allows this behavior to be implemented easily by the server.

The application developer can add framework or utility classes and provide additional implementations as well as change some HP OODCE default behavior. Additionally, the developer continues to have access to the C language-based DCE API. Direct use of this API is governed only by the correct mapping of exceptions and the corresponding rules for C++ with regard to the C language.

HP OODCE Exception Model

One goal of the HP OODCE system was to create a consistent error model. C++ exception handling was the natural choice as the basis for this model since this mechanism is already well integrated into the language. C++ provides benefits such as object destruction and reduced source code size and is similar in principle to the current DCE exception handling mechanism.

Despite their similarity, the C++ and DCE exception mechanisms do not integrate well. Exceptions raised by one implementation cannot be caught by the other, and more important, those generated by the DCE implementation can cause memory leaks if they are allowed to propagate through C++ code. This latter problem is a result of the use of the setjmp and longjmp functions in the DCE exception implementation, which do not allow run-time C++ to call destructors for temporary and explicitly declared objects before exiting a particular scope.

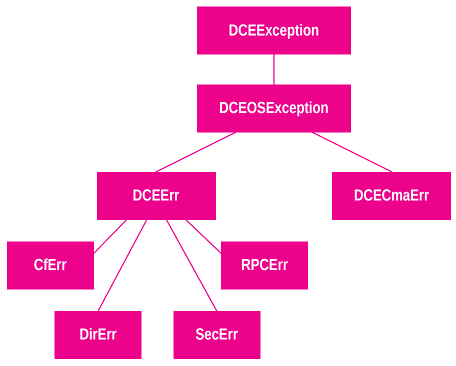


Fig. 10. Exception class hierarchy.

* uuid_t is a C structure containing all the characteristics for a UUID.

** Mutexes, or mutual exclusion locks, are used to protect critical regions of code in DCE threads.

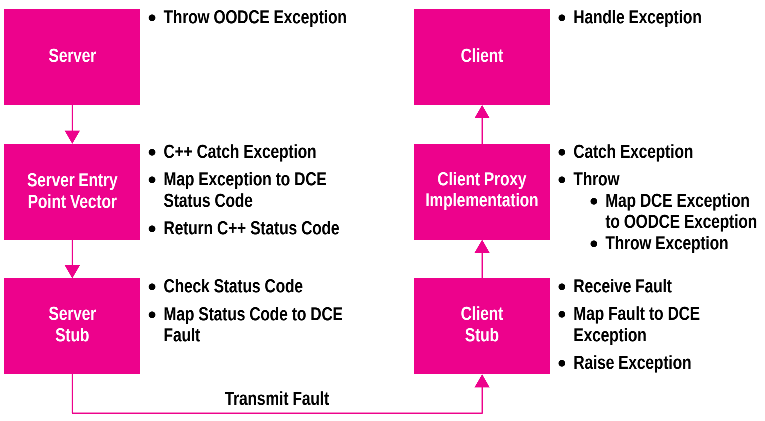


Fig. 11. Exception handling in HP OODCE.

To solve the problems raised by the use of two different exception mechanisms, HP OODCE maps DCE exceptions into C++ exceptions. The HP OODCE classes are arranged into a C++ class hierarchy (see Fig. 10). DCEException is the base class for the hierarchy and provides pure virtual operators to convert exceptions to status codes or ASCII strings. The hierarchy contains subclasses derived from the base class for each of the DCE subcomponents (RPC, security, directory services, configuration, CMA (common multi-threaded architecture) threads, and so on) so that each individual DCE exception can be caught by type.

HP OODCE takes particular care to prevent DCE exceptions from being propagated directly into C++ code. At the boundaries between DCE C and HP OODCE C++ code, DCE exceptions and error status codes are mapped into HP OODCE exceptions and propagated into C++ code. One area that needed particular attention was in passing exceptions between the server and client. We wanted to use the RPC run-time implementation of the server's communication fault transmission, but to do so required a "translation" layer to isolate RPC exceptions from HP OODCE C++ code. This translation layer is implemented within the idl++-generated client proxy implementation and server entry point vector classes (see Fig. 11). C++ exceptions raised in the HP OODCE server are caught in the server entry point vector and mapped to a DCE status code. This status code is then returned to the server stub, which translates the code into a DCE exception and raises it to the attention of the run-time RPC. The run-time RPC takes care of mapping the exception to one of the currently implemented RPC fault codes and then transmits the fault to the client. Basically the reverse happens on the client side, except that here, the client implementation class will catch the DCE exception raised from the client stub and throw the HP OODCE exception back to the client.

Acknowledgments

The idea, overall architecture, initial design, and prototype implementation of this project are mainly attributed to Jeff Morgan. Deborah Caswell, John Dilley, and Bob Fraley each designed and implemented significant parts of the system. We wish to thank them for their continuing advice and contributions to this product. We also wish to thank Amy Arnold, Serena Chang, Jack Danahy, Linda Donovan, John Griffith, and John Stodieck for their effort in making HP OODCE an HP product.

Bibliography

1. J. Dilley, "OODCE: A C++ Framework for the OSF Distributed Computing Environment," *Proceedings of the Winter '95 USENIX Conference*.

2. Open Software Foundation, *OSF DCE Application Environment Specification*, 1992.

3. M.Ellis and B.Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, May, 1991.

Glossary

Although the terminology associated with object-oriented programming and C++ has become reasonably standardized, some object-oriented terms may be slightly different depending on the implementation. Therefore, brief definitions of some of the terminology used in this paper are given below. For more information on these terms see the references in the accompanying article.

Abstract Class. Abstract classes represent the interface to more than one implementation of a common, usually complicated concept. Because an abstract class is a base class to more than one derived class, it must contain at least one pure virtual function. Objects of this type can only be created through derivation in which the pure virtual function implementation is filled in by the derived classes.

The following is an example of an abstract base class:

```
class polygon {
public:
      // constructor, destructor and other member functions
     // could go here...
     virtual void rotate (int i)  =  O; //a pure virtual function
     // other functions go here...
};
```

Other classes, such as square, triangle, and trapezoid, can be derived from polygon, and the rotate function can be filled in and defined in any of these derived classes.

Base Class. To reuse the member functions and member data structures of an existing class, C++ provides a technique called class derivation in which a new class can derive the functions and data representation from an old class. The old class is referred to as a base class since it is a foundation (or base) for other classes, and the new class is called a derived class. Equivalent terminology refers to the base class as the superclass and the derived class as the subclass.

Catch Block. One (or more) catch statements follow a try block and provide exception-handling code to be executed when one or more exceptions are thrown. Caught exceptions can be rethrown via another throw statement within the catch block.

Class. A class is a user-defined type that specifies the type and structure of the information needed to create an object (or instance) of the class.

Concrete Data Class. Concrete data classes are the representation of new user-defined data types. These user-defined data types supplement the C++ built-in data types such as integers and characters to provide new atomic building blocks for a C++ program. All the operations (i.e., member functions) essential for the support of a user-defined data type are provided in the concrete class definition. For example, types such as complex, date, and character strings could all be concrete data types which (by definition) could be used as building blocks to create objects in the user's application.

The following code shows portions of a concrete class called date, which is responsible for constructing the basic data structure for the object date.

```
typedef boolean int;
#define TRUE 1
#define FALSE O

class date {
public:
      date (int month, int day, int year); //Constructor
      ~date(l;                                   //Destructor
      boolean set date(int month, int day, int year);
      // Additional member functions could go here. . .

private
      int year;
      int numerical_date;
      // Additional data members could go here...
};
```

Constructors. A constructor creates an object, performing initialization on both stack-based and free-storage allocated objects. Constructors can be overloaded, but they cannot be virtual or static. C++ constructors cannot specify a return type, not even void.

Derived Class. A class that is derived from one (or more) base classes.

Destructors. A destructor effectively turns an object back into raw memory. A destructor takes no arguments, and no return type can be specified (not even void). However, destructors can be virtual.

Exception Handling. Exception handling in C++ provides language support for synchronous event handling. The C++ exception handling mechanism is supported by the throw statement, try blocks, and catch blocks.

Member Functions. Member functions are associated with a specific object of a class. That is, they operate on the data members of an object. Member functions are always declared within a class declaration. Member functions are sometimes referred to as methods.

Object. Objects are created from a particular class definition and many objects can be associated with a particular class. The objects associated with a class are sometimes called instances of the class. Each object is an independent object with its own data and state. However, an object has the same data structure (but each object has its own copy of the data) and shares the same member functions as all other objects of the same class and exhibits similar behavior. For example, all the objects of a class that draws circles will draw circles when requested to do so, but because of differences in the data in each object's data structures, the circles may be drawn in different sizes, colors, and locations depending on the state of the data members for that particular object.

Throw Statement. A throw statement is part of the C++ exception handling mechanism. A throw statement transfers control from the point of the program anomaly to an exception handler. The exception handler catches the exception. A throw statement takes place from within a try block, or from a function in the try block.

Try Block. A try block defines a section of code in which an exception may be thrown. A try block is always followed by one or more catch statements. Exceptions may also be thrown by functions called within the try block.

Virtual Functions. A virtual function enables the programmer to declare member functions in a base class that can be redefined by each derived class. Virtual functions provide dynamic (i.e., run-time) binding depending on the type of object.

HP Encina/9000: Middleware for Constructing Transaction Processing Applications

A transaction processing monitor for distributed transaction processing applications maintains the ACID (atomicity, consistency, isolation, and durability) properties of the transactions and provides recovery facilities for aborting transactions and recovering from system or network failures.

by Pankaj Gupta

Transaction processing systems are widely used by enterprises to support mission-critical applications, such as airline reservation systems and banking applications. These applications need to store and update data reliably, provide concurrent access to the data by hundreds or thousands of users, and maintain the reliability of the data in the presence of failures.

The HP Encina/9000 transaction processing monitor [1] provides the middleware for running transaction processing applications. It maintains ACID (atomicity, consistency, isolation, and durability) properties for transactions (see the glossary on page 65). It ensures that applications that run concurrently will maintain data consistency. Encina/9000 also provides recovery facilities for aborting transactions and recovering from failures such as machine or network crashes.

DCE and Distribution

Encina/9000 provides the ability to write distributed applications. Encina/9000 applications can be written as client/server applications with the client and server possibly running on different machines. Encina/9000 servers can communicate and cooperate with each other in updating data on several different machines.

Distributed applications provide several advantages. The data maintained by an enterprise may itself be distributed because of historical and geographical considerations. Furthermore, distributed applications are able to exploit parallelism by running concurrently on several machines.

Distributed computing offers the advantage of improved performance, availability, and access to distributed data. Performance is improved by spreading the computing among various machines. For example, the application's user interface can be run on a PC while the user code could be split to run on several machines. The use of multiprocessing machines to provide parallelism for multiple users can improve the throughput of the system. Availability can be increased by a distributed system in which replication is used to keep several copies of the data. Access to distributed data or to data that is maintained in several databases is also facilitated by distributed computing.

Data may be distributed because the database becomes too large or the CPU on the database machine becomes a bottleneck. Data can also be distributed to increase availability and improve the response time by keeping the data close to the users accessing it. Finally, data can be distributed to keep separate administrative domains, such as different divisions in a corporation that want to keep their data local.

Encina/9000 uses the Open Software Foundation's DCE [2] (Distributed Computing Environment) as the underlying mechanism for providing distribution. It uses the DCE RPC mechanism to provide client/server communication. Encina/9000 is also very closely tied to DCE naming and security services (see the articles on pages 28 and 41 for more about these services). For example, an Encina/9000 server can be protected from unauthorized use by defining access control lists (ACLs). ACLs contain an encoding of the authorization policy for different users and are enforced by DCE at run time. ACLs are described on page 49. Encina/9000 also makes use of the threading package provided by DCE.

To achieve optimum price and performance, careful consideration needs to be given to how the data and the application are partitioned. Throughput and response times are often the key criteria by which users judge the performance of a system. Encina/9000 provides the flexibility of being able to specify the distribution topology of the application. In addition, users can specify data replication if it will help to ensure higher availability of mission-critical data.

Two-Tiered versus Three-Tiered Architectures

In the past, transaction processing applications were implemented using a two-tiered architecture (see Fig. 1). In this

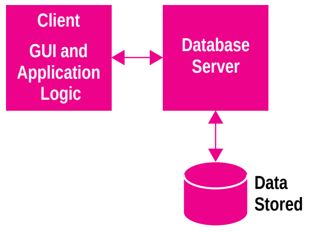


Fig. 1. Two-tiered architecture for transaction processing.

paradigm an application is written as a client, which accesses a database server. The client implements the graphical user interface (GUI) and the application logic. The database server handles access to the data stored in a database.

The advantage of this approach is simplicity. The disadvantage is that it is not scalable beyond a certain point. It is also less flexible and harder to modify to meet new business needs.

Encina/9000 allows the development of applications using a three-tiered architecture like the one shown in Fig. 2. In this paradigm, an application is partitioned into a client that implements the graphical user interface to the user, an application server that implements the business logic of the application, and a database server that implements the database access.

The Encina/9000 three-tiered architecture provides the following advantages over traditional two-tiered architectures:

- Decoupling the GUI from the business logic
- Scalability of the architecture to support a very large number of users and a high transaction throughput
- Accessibility to multiple database servers from an application server
- Freedom from being tied into any particular database vendor
- Tight integration with the distributed computing facilities offered by DCE
- Choice of transactional applications that support any combination of RPC, CPI-C (Common Programming Interface for Communications), and queued message communication
- Ability to retain data on a mainframe or other legacy computer and reengineer by adding HP-UX* application servers, providing lower cost and higher price/performance relative to some mainframe systems.

Three-tiered architectures are more complicated in general but provide greater flexibility of application design and development. Among the reasons why users are willing to give up the simplicity of the two-tiered architecture is the faster response times and the more effective user interfaces provided by the three-tiered architecture. The ability to provide a front-end workstation that supports graphical user interfaces gives an application a more effective user interface. For applications that require access to data distributed across large geographic regions, a three-tiered architecture offers more flexibility to tune the communications to compensate for WAN delays and improve availability. This results in a faster response time because the user is accessing local data most of the time. Propagation of the data to other machines can be queued and performed offline. Therefore, geographically distributed data can be maintained without having to perform expensive distributed two-phase commit protocols online.

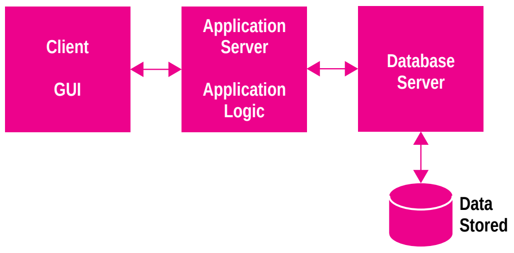


Fig. 2. Three-tiered architecture for transaction processing.

Two-phase commits that happen over wide area networks are expensive and care must be taken when designing distributed applications to minimize the amount of two-phase commits over the network. Queued communications also improve availability. See page 65 for a definition of commit.

Components of Encina/9000

Fig. 3 shows the architecture for the implementation of Encina/9000 that runs on the HP-UX* operating system.

Each of the components shown in Fig. 3 is packaged independently. A machine that runs Encina/9000 clients only can be configured without the Encina/9000 server software. Machines that run Encina/9000 servers must be configured with both the Encina/9000 client and server components.

Encina/9000 applications that can be configured to run on top of the Encina/9000 server component include:

- Peer-to-peer communication, which provides transactional access to data stored on mainframes and workstations running the HP-UX operating system
- Structured file system, which is a record-oriented file system base on the X/Open® ISAM (index sequential access method) standard
- Recoverable queueing service, which provides applications with the ability to enqueue and dequeue data
- Monitor, which is an infrastructure for application development, run-time support, and administration.

The DCE components used by Encina/9000 include: RPC, the directory service, the security service, and threads.

Encina/9000 Toolkit

The Encina/9000 client component is also called the Encina/9000 toolkit executive and the Encina/9000 server component is also called the Encina/9000 toolkit server core. Together these components are called the Encina/9000 toolkit.

Fig. 4 shows the components that make up and support the Encina/9000 toolkit.

Base Development Environment. The lowest layer of the Encina/9000 toolkit is the base development environment. It provides developers of other Encina/9000 components with a uniform set of features independent of the underlying operating system. The base development environment library

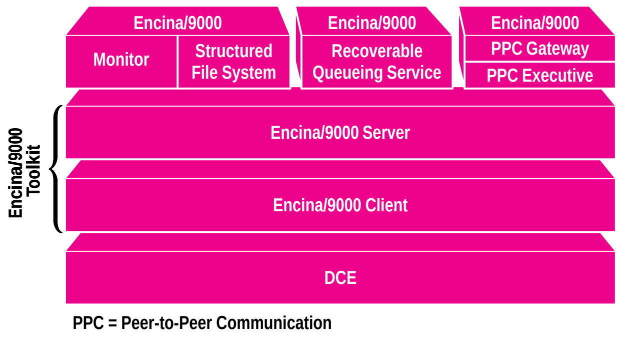


Fig. 3. The architecture of Encina/9000 on the HP-UX operating system.

provides a common platform independent threading interface and an abstraction for low-level functions so that the upper layers that use the base development environment can be independent of differences in the operating system or the hardware platform on which Encina/9000 runs.

The base development environment provides support for multiple threads of execution by using DCE threads. Also, it provides thread-safe routines for the following functionality:

- Memory management
- File I/O
- Process management
- Signal handling
- Timers and alarms
- Native language support.

The base development environment is intended primarily for the development of other Encina/9000 components.

Transaction Manager. The transaction manager provides the ability to demarcate transactions, which means that it is able to specify the beginning, the commit, and the abort of a transaction. Internally it supports a distributed two-phase commit management protocol, including the ability to perform coordinator migration.

The transaction manager supports nested transactions capability,[3] which allows nested transactions to be defined within a top-level transaction. Nested transactions have isolation and durability properties similar to regular transactions, but the abort of a nested transaction does not cause the top-level transaction to abort. This allows a finer granularity of failure isolation in which the main transaction can handle the failure of certain components implemented with a nested transaction. Nested transactions are defined in the glossary on page 65.

The application must be carefully designed since failures such as crashed server nodes, which cause a nested transaction to fail, could in some cases also cause the top-level transaction to fail. The Encina/9000 structured file system provides support for nested transactions for data stored in structured files. However, database vendors like Oracle do not currently support nested transactions in their products, making it impossible to exploit the advantages of Encina/9000's nested transaction capabilities for data stored by these relational databases.

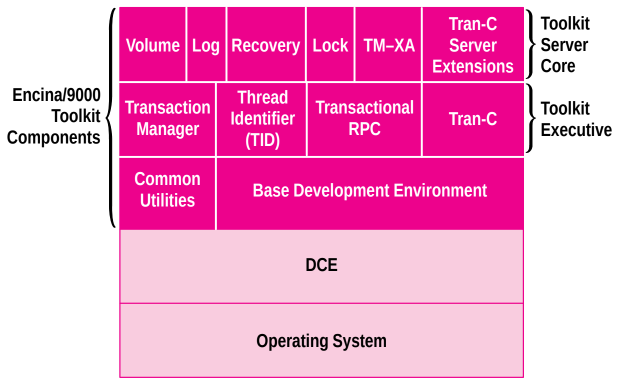


Fig. 4. A detailed view of the components that make up and support the Encina/9000 toolkit.

The Encina/9000 transaction manager provides an application program with the ability to issue callbacks on events related to the transaction's commit protocol. This enables the programmer to write routines that are invoked before the transaction prepares or aborts or after the coordinator decides to abort or commit the transaction.

The transaction manager allows transactions to be heuristically committed by a system administrator. This should only be used in rare cases in which the transaction coordinator is unavailable and the administrator does not want to block access to locked data and has to trade off data availability to avoid possible data inconsistency.

Thread-to-TID. Since Encina/9000 makes use of DCE threads, the work done on behalf of a user transaction can be composed of several different threads. The thread-to-TID service associates a transaction with a thread and maintains the mapping between a thread and a transaction identifier (TID). This service is used by other Encina/9000 components and is rarely used by programmers directly.

Transactional RPC. The Encina/9000 transactional RPC service enhances the DCE RPC mechanism to provide transactional semantics for remote procedure calls. Unlike remote procedure calls, transactional RPCs have once-only semantics. If a transaction performing an RPC commits, then the RPC is guaranteed to have executed once and once only. If the transaction performing an RPC aborts then the RPC is guaranteed not to have executed (if the RPC was executed its effects are undone by the transaction abort).

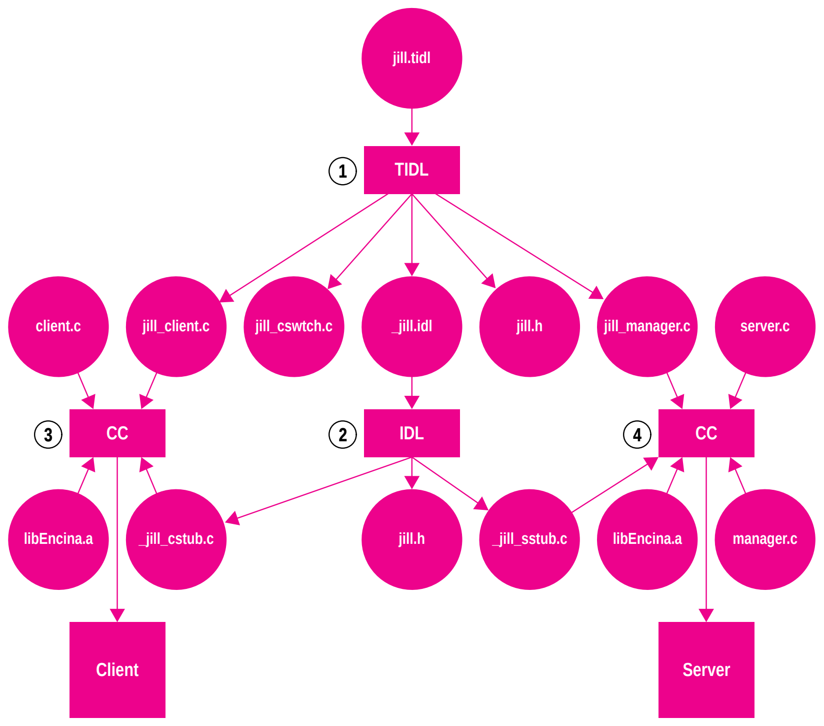
A transaction can make transactional RPC calls to multiple servers, and a server can in turn make a transactional RPC call to another server.

The transactional RPC service extends the DCE RPC model. The interface definition for the service executed on behalf of a transaction is defined in a TIDL (Transactional Interface Definition Language) file, which is similar to a DCE IDL file.* This file must be preprocessed with a TIDL compiler (similar to an IDL compiler). The TIDL preprocessor generates client stubs, server stubs, a header file, and an IDL file. The DCE IDL preprocessor is run on the IDL file to generate additional stubs and header files. The client and the server executables are generated by compiling and linking the various stub sources and libraries. This process is illustrated in Fig. 5.

Transactional RPC also supports nontransactional RPCs (a nontransactional RPC call can be made by calling the transactional RPC service). The TIDL file for the service interface must specify that the service is nontransactional.

Log. This component of Encina/9000 provides logging capabilities. It provides write-ahead logging (see glossary) for storing log records that correspond to updates to recoverable data and log records that correspond to transaction outcomes. The log records are used by the transaction manager to undo the effects of transactions that have aborted and to ensure that the committed transactions are durable.

* See the article on page 55 for more about IDL files.



1. Preprocess transaction through the TIDL compiler.
2. Run the IDL file created in 1 through the IDL compiler.
3. Create the client program.
4. Create the server program.

Fig. 5. The steps involved in turning a transaction into client and server executables.

For earlier versions of Encina/9000, the log service was implemented to provide a log server that could be used by many different clients to store log records. The latest version of Encina/9000 supports the log service as a library which is linked into the client code.

The log service supports archiving of log data for crash and media recovery. It also supports mirroring of data.

Lock. This component provides two-phase locking (see glossary) facilities to ensure the isolation and consistency properties of transactions. Applications can request locks on resources before accessing them, and the lock manager ensures that the lock request on behalf of a transaction will not be granted if another transaction holds a conflicting lock on that resource. Locks are released automatically when the transaction completes, and the application may also request early release of locks when it is safe to do so. The lock service also supports locking for nested transactions.

The locking service implements logical locking in which the programmer defines lock names and associates the lock names with physical resources. When a programmer wants to lock a physical resource, the logical lock name associated with that resource is specified in the call to the locking service.

In addition to supporting the conventional read/write locks, Encina/9000 also supports intention locks and instant duration locks. Intention locks are used to declare an intent to subsequently lock a resource. The use of intention locks can reduce the potential for deadlock among concurrent transactions. Instant duration locks are locks that are granted but not held and can be used to implement complex locking algorithms.

The lock service also provides the ability to determine if a transaction is deadlocked or not.

Volume. This component maintains the data storage in terms of logical data volumes. It provides the ability to manage very large files and view multiple physical disks as a virtual file. It also supports the ability to mirror a data volume transparently to the client. The volume component sometimes sacrifices speed for increased reliability and may be inappropriate for certain applications. This component is currently not used by the log component.

TM-XA. The Encina/9000 TM-XA component implements the X/Open XA interface. The XA interface is a bidirectional interface between a transaction manager and a resource manager such as a database. The XA interface provides a standard way for transaction managers to connect to databases.

The use of TM-XA with the Encina/9000 monitor is recommended. In this case the server registers each resource manager with a call providing the name of the resource manager and an associated switch structure,[4] which gives the Encina/9000 TM-XA component information about the resource manager. In addition, the server must also be declared as a recoverable server. TM-XA allows Encina/9000 to hook up with standard database products such as Oracle, Informix, and so on.

Glossary

The following are brief definitions of some of the transaction-related terminology used in the accompanying article.

Transactions and ACID

A transaction is the logical grouping of a user function performed as a unit so that it maintains its ACID (atomicity, consistency, isolation, and durability) properties. Transactions allow users to execute their programs and modify shared resources like databases in the presence of simultaneous access and updates by multiple users and in the presence of various kinds of failures.

Atomicity of a transaction means that either all the actions specified within the transaction will be performed or none of them will be performed. This ensures that a transaction is not partially applied, which is desirable since a partial application of the user transaction could leave the database in an inconsistent state. Consistency means that the database consistency is preserved in the presence of concurrency among multiple users. Isolation means that while the transaction is executing, its effects will not be visible to other concurrently running transactions. Durability means that once a transaction has been successfully completed the effects of that transaction are made permanent and survive failures.

Commit, Abort, and Prepare

A successful completion of a transaction is called a *commit* of the transaction. Before a transaction commits, it can be aborted either by the user or by the transaction processing system.

A user organizes a set of actions in a transaction with the intent that all of these actions should happen or none of these actions should happen. An example of such a transaction is a transfer of money between a person's savings account and checking account. This transaction consists of the actions of changing the savings account balance and the checking account balance. If the transaction is successful then both these account balances should be changed, one debited by amount X, and the other credited by amount X. Any other outcome would be in error. When the user submits this transaction and all operations in the transaction are successfully carried out, the transaction is said to have committed.

It may not always be possible to commit a transaction. For example, the machine that maintains the checking account balance may be down, or the user may have supplied an incorrect personal identification number (PIN) or decided to cancel the request after submitting it. If the transaction cannot be performed, then it is said to have been aborted or simply to have aborted. If a transaction is aborted (aborts), then none of the actions of the transaction are performed (or if they had been performed they are undone).

Transaction processing systems use mechanisms like two-phase locking to ensure the isolation properties, and *two-phase commit* protocols to ensure that all the participants within a transaction can be atomically committed or aborted.[1]

A two-phase commit protocol is typically used to commit a transaction in which multiple participants are performing actions requested by the transaction. One of these participants is called the coordinator. In the first phase of the two-phase commit protocol, the coordinator sends a *prepare* message to all the participants, asking them to prepare the transaction and send a message back indicating whether or not they can prepare the transaction. If all the participants respond that they can prepare the transaction, the coordinator writes a record in its log indicating that the transaction is committed and instructs all the participants to commit the transaction (this is the second phase of the protocol). If any participant responds back to the coordinator that it is unable to prepare the transaction, the coordinator notes in its log that the transaction is aborted and instructs all participants to abort the transaction.

When a participant receives a commit message from the coordinator in the second phase, it must ensure that all the actions of the transaction are durably stored on disk. When a participant receives an abort message from the coordinator in the second phase, it must ensure that all the actions of the transaction are undone. Therefore, in the first phase of the two-phase commit protocol, the participant must ensure that data is stored reliably in logs that enable it subsequently to undo the effects of a transaction or make permanent the effects of a transaction.

Nested Transaction

In the nested transaction model, a transaction (also called a top-level transaction) can be decomposed into a tree-like hierarchy. For example, a debit-credit transaction can be decomposed into two subtransactions, one for debit and one for credit. The debit or credit subtransactions could be further decomposed into smaller subtransactions. A subtransaction maintains the durability and consistency properties of its parents. The difference is that a subtransaction can be aborted and reexecuted without aborting its parent. In the debit-credit example, the debit subtransaction can be aborted and reexecuted without having to abort the entire transaction. In this case the top-level transaction verifies that each subtransaction can be committed at the time of transaction commit.

Two-Phase Locking

Two-phase locking means that a transaction will have two phases. In the first phase the transaction can only acquire and not release locks, and in the second phase it can only release and not acquire locks.

Write-Ahead Logging

For data recovery, transaction processing systems use a log to log data that is being modified. In write-ahead logging, data is copied to the log before it is overwritten. This ensures that if the transaction is aborted, the data can be restored to its original state.

Reference

1. J. Gray and A. Reuters, *Transaction Processing Concepts and Techniques*, Morgan Kaufman, 1993.

TM-XA allows an Encina/9000 application to make calls to one or more resource managers that support the XA interface. The Encina/9000 application starts a transaction, accesses a resource manager using its native SQL interface, and then commits the transaction. The TM-XA software coordinates the commit of the transaction among the various resource managers and other Encina/ 9000 components (like the structured file system or the recoverable queuing system) which might be accessed by the user transaction.

Recovery. This component drives the recovery protocols to recover from failures. It provides recoverable memory management. Recovery also provides the ability to perform:
- Abort recovery. This ensures that after a transaction is aborted, it is rolled back at all participating sites.
- Crash recovery. This provides a recovery after a system failure by rolling back all the transactions that had not been committed and rolling forward all the committed transactions.
- Media recovery. This is used to provide recovery when data written to the disk is destroyed.

The recovery component produces and uses the records written by the log service and ensures the consistency of transactional data. In the case of a transaction failure, the recovery component undoes the effects of a transaction. During recovery from a system failure, this component will examine the records in the log, appropriately commit or abort transactions for which it finds records in the log, and bring the data to a consistent state.

For media failures, the system administrator must provide archives that are used by the recovery component to restore the data to the state it was in when the archive was created.

There may be a loss of data in the case of media recovery. Encina/9000 also provides the ability to perform online backups to create the media archives necessary for recovery.

Tran-C

Encina/9000 provides extensions to the C programming language to make it easy to invoke the functionality provided by the Encina/9000 toolkit. Tran-C consists of library functions and macros that provide a simple programming paradigm so that the user does not have to access the toolkit module interfaces directly. The user can invoke high-level Tran-C constructs rather than the lower-level toolkit calls. The use of Tran-C versus toolkit calls is analogous to using a high-level language versus assembly language. Using the toolkit primitives directly is much more flexible, but the flexibility comes at the price of far greater complexity. In general, Tran-C is recommended for application programming.

The most important constructs provided by Tran-C are the transaction, onCommit, and onAbort clauses. These constructs provide a mechanism for the programmer to start a transaction and declare code to execute when the transaction commits or aborts. This is illustrated in Fig. 6. The application programmer is freed from the task of initializing all the underlying toolkit components and manually managing transaction identifiers, transactional locks, and other transactional metadata. All the code bracketed by the transaction clause is executed on behalf of the same transaction. When a transaction bounded by the transaction construct aborts (or commits), control in the program automatically transfers to the associated onAbort (or onCommit) clause.

Tran-C also supports nested transactions and multiple threads of control. The concurrent and cofor constructs can be used to create multiple concurrent threads within a transaction. The concurrent construct is used to enable an application to concurrently execute a predetermined number of threads, while the cofor construct enables the application to concurrently execute a variable number of threads. Both constructs provide the ability to create multiple threads which can be run either as subtransactions or as concurrent threads within the transaction. The subTran construct allows a created thread to be executed as a subtransaction within the parent transaction. The subThread construct allows a created thread to be executed as a separate thread within a nested transaction.

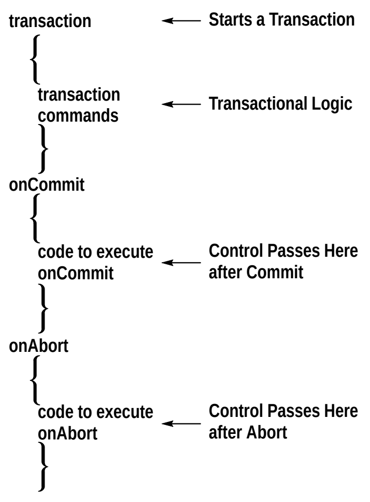


Fig. 6. A code fragment illustrating the use of the Tran-C constructs Transaction, onCommit, and onAbort.

Toolkit Example

Fig. 7 shows an example of the interactions between the components of the Encina/9000 toolkit. In this example a client makes a call to update data stored by a database and then commits the transaction. The following steps are associated with the circled numbers in Fig. 7.

1. The client starts a transaction by making a call to the transaction manager.
2. The client performs a transactional RPC by making a call to the transactional RPC component.
3. The transactional RPC component makes a call to the transaction manager to obtain transactional data for the transaction.
4. The transactional RPC component calls DCE RPC to transmit the user data and transactional data to the server.
5. DCE RPC (on the server) makes a call to the transactional RPC.
6. The transactional RPC component passes the transactional information to the transaction manager.

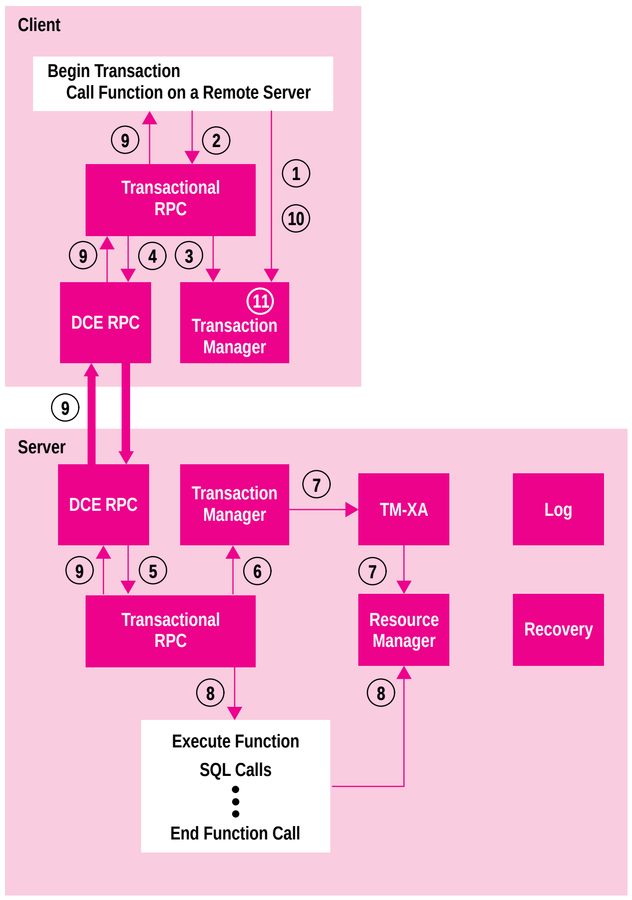


Fig. 7. An example of the interactions between components of the Encina/9000 toolkit.

7. The transaction manager uses the TM-XA interface to call the resource manager.
8. The transactional RPC component calls the user function that makes SQL calls to the resource manager. The resource manager performs the appropriate locking and updating of its data.
9. The user function on the server returns to that transactional RPC component which then returns to the client via DCE.
10. The client calls the transaction manager to commit the transaction.
11. The transaction manager uses a two-phase commit protocol to commit the transaction. It contacts all the transaction manager participants that have participated in the transaction. Each transaction manager uses the recovery and log components to log the prepare and commit decisions during various phases of the commit protocol for the transaction.

Peer-to-Peer Communications

Encina/9000 peer-to-peer communications, or PPC, provides transactional access to data stored on mainframes, and it performs a distributed two-phase commit of data stored on mainframes and HP 9000 servers. This allows mainframe applications to participate in an Encina/9000 transactional application, and conversely, an Encina/9000 application is able to participate in a mainframe transactional application. Encina/9000 PPC uses a two-phase commit sync protocol (sync level 2) to commit a transaction that accesses data on a mainframe and an HP 9000 server.

PPC services are implemented as a PPC executive and a PPC gateway product. These products can be purchased separately. The PPC executive is a library that runs in a DCE cell, and the PPC gateway is a server that acts as a gateway between DCE and SNA communications protocol. This gateway allows Encina/9000 applications to communicate with LU 6.2 applications.*

A typical PPC configuration involves an Encina/9000 PPC application running in a DCE cell and communicating with a PPC gateway server running in the same DCE cell. The PPC gateway server communicates with the mainframe using an SNA communications package. PPC provides the ability to write Encina/9000 applications that act as either the coordinator or the subordinate in a transaction between an Encina/9000 system and a mainframe host. Encina/9000 application programmers use the CPI-C API for coding the PPC component. The PPC gateway translates the CPI-C conversations from TCP/IP to LU6.2. This is illustrated in Fig. 8.

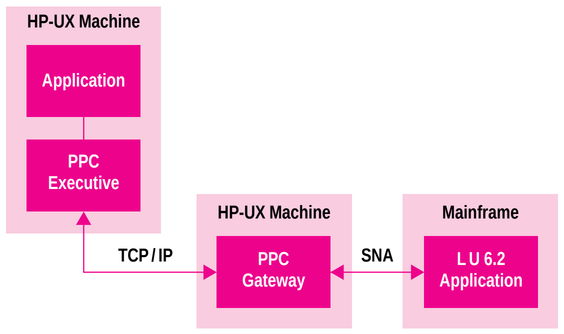


Fig. 8. A PPC configuration showing the PPC gateway translating TCP/IP protocol to SNA protocol.

Structured File System

The structured file system is a record-oriented file system based on the X/Open ISAM standard. It provides an alternative to other commercial resource managers and the ability to support nested transactions that access data in the structured file system. It also provides full transactional integrity.

The records in the structured file system contain different fields that can be indexed by primary keys and secondary keys. The structured file system's field and record types are similar to those used by the recoverable queuing service (described below), allowing applications to have easy access to both systems. In addition, the structured file system supports a COBOL interface with the structured file system's external file handler.

Files in the structured file system are organized in one of the following three ways: entry-sequenced, relative, and B-tree clustered (see Fig. 9). Records in an entry-sequenced file are stored in the order in which they are written into the file. New records are always appended to the end of the file. A relative file is an array of fixed-length slots. Records can be inserted in the first free slot found from the beginning of the file, at the end of the file, or in a specified slot in the file. A B-tree clustered file is a tree-structured file in which records with adjacent index names are clustered together.

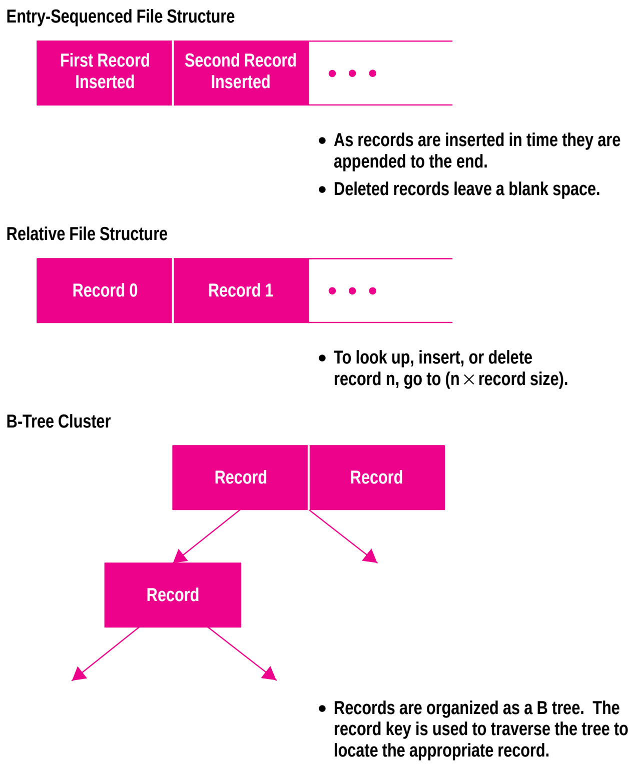

Entry-Sequenced File Structure
First Record Inserted
Second Record Inserted
• • •
• As records are inserted in time they are appended to the end.
• Deleted records leave a blank space.
Relative File Structure
Record 0
Record 1
• • •
• To look up, insert, or delete record n, go to (n × record size).
B-Tree Cluster
Record
Record
Record
• Records are organized as a B tree. The record key is used to traverse the tree to locate the appropriate record.

Fig. 9. File organizations supported in the structured file system.

* LU 6.2 applications are mainframe applications that are written to run on top of IBM's LU 6.2 protocol.

The structured file system is simple and fast, but limited in flexibility when compared to relational databases. Relational databases provide powerful and complex access semantics with operations such as select, join, aggregate, and so on. The structured file system provides low-level access to records whose formats are user-defined and controlled.

Recoverable Queueing Service

Encina/9000 provides a recoverable queueing service which is layered on top of the basic toolkit and server core components. This service provides applications with the ability to transactionally queue and dequeue data. Application developers can write applications that transactionally update data in a resource manager like a database and queue or dequeue data with the guarantee that either both operations will succeed or both operations will abort.

An example of a transactionally recoverable queue would be a banking application that sends a letter to a customer if the customer's balance goes below zero. The action to generate the letter can be queued and processed later at the end of the day. The recoverable queue ensures that this action will always be performed even in the event of system failures.

One advantage of the queueing model is that applications can offload some work to be done at a later time. This deferred mode of computing is in contrast with the RPC style of communication in which an application invokes a service to do the processing as soon as it can.

A queue is a linear data structure. Elements in the data structure are queued in a particular configurable order and the dequeue occurs on a FIFO (first in, first out) basis. An element of a recoverable queueing service queue is structured in a record-oriented format. Encina/9000 supports queues that may contain elements of different data types. An element key is a sequence of one or more fields of an element type that are used to retrieve an element.

Encina/9000 provides the ability to define one or more recoverable queueing service servers in an Encina/9000 cell. Each server can internally support multiple queue sets. A queue set is a collection of queues within a recoverable queueing service server. Applications can queue or dequeue to or from a particular queue set. Queues within a queue set can be assigned priority classes relative to each other. Also, service levels define how to distribute the dequeues so that the queues with lower priority are not starved.

The recoverable queueing service supports a weak FIFO locking behavior. For example, when two transactions concurrently dequeue from a queue, each obtains a lock on the first element that it can lock on the queue. It is possible for the transaction that obtained a lock on the second element in the queue to commit before the transaction that obtained a lock on the first element in the queue. Another consequence of the weak FIFO locking policy may be that a transaction that consecutively queues multiple elements may not be able to place all these elements in that queue in an uninterrupted sequence.

The recoverable queueing service uses the DCE security mechanisms to secure access to the queue. Administratively, ACLs (access control lists) can be set up to authorize users or groups to be able to perform queue operations like read from queue, queue to the queue, dequeue from the queue, delete a queue, and so on.

A recoverable queueing service queue can be scanned using element keys, cursors (for sequential access), or element identifiers.

Finally, the recoverable queueing service provides the ability to register callbacks with the service's server on callback quantities such as the number of elements, size in bytes, and work accumulation. For example, with this feature it is possible to write applications that can ask the recoverable queueing service server to inform them when ten elements have been queued.

Monitor

The Encina/9000 transaction processing monitor provides an infrastructure for application development, run-time support, and administration. It supports the development of a three-tiered architecture in which multiple clients can access data stored in multiple resource managers.

Like DCE, the Encina/9000 monitor also has the concept of a cell. For the Encina/9000 monitor the cell is called an *Encina cell*. The Encina cell is a subset of the DCE cell, and multiple Encina cells can be defined within a DCE cell. (DCE cells are described in the article on page 6.) An Encina cell may consist of multiple nodes. A node is either a public node or a secure node. A secure node is a node on which the Encina/9000 servers can be securely run. Public nodes are nodes where only clients are run. Servers are not configured to run on public nodes. An example of an Encina cell is shown in Fig. 10. Like DCE, an Encina cell has a cell administrator who is responsible for performing administrative tasks.

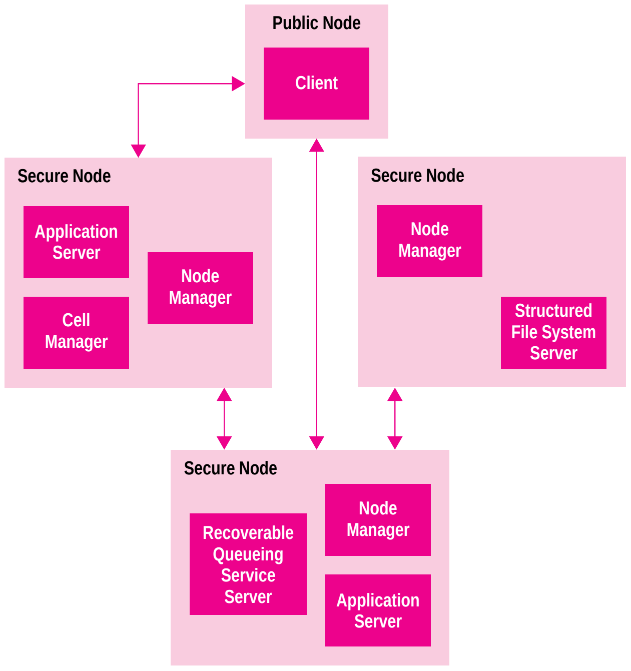


Fig. 10. The components in an Encina/9000 cell.

The encina cell contains the following server processes:

Cell Manager. There is one cell manager process in an Encina cell. The cell manager maintains the data needed to configure and administer the Encina cell. This data is stored in a data repository managed by the structured file system. The data describes how the application servers are configured to run on the secure nodes and includes the authorization information for those servers. The cell manager also monitors the state of the node managers and keeps statistics on the use of the servers by the clients.

Node Manager. There is one node manager process in each secure node in an Encina cell. The node manager monitors the application servers that are running in that node. If an application server fails, the failure is detected by the node manager which then restarts the application server.

Application Server. The server part of a user application is an application server. Typically, application servers accept calls from an Encina cell's clients and then process the user requests by accessing one or more resource managers. Application servers may be recoverable or ephemeral. A recoverable application server is one that uses the underlying Encina/9000 facilities to provide the ACID (atomicity, consistency, isolation, and durability) transactional properties. When a recoverable server fails, it performs recovery on restart which guarantees the consistency of the data. An ephemeral server does not provide the ACID properties to the data it accesses. An application server consists of a scheduling daemon process (called mond) and one or more processes (called PAs) that accept client requests. PAs are multithreaded processes. The mond coordinates the clients' requests for servers and assigns a PA to a requesting client. In this respect the role of the mond is similar to that of the RPC daemon rpcd in DCE. The mond also monitors the PAs and automatically restarts a PA in the event that the PA dies. An example of an application running in an Encina/9000 cell is shown in Fig. 11.

In the Encina/9000 monitor environment, a client can make a server request using explicit binding or transparent binding. With transparent binding the client simply makes a call to the server and the monitor environment is responsible for routing the client request to an appropriate server. With explicit binding, a client explicitly binds to a particular server. The Encina/9000 monitor provides a call to request a list of all servers exporting a particular interface, a call to get a handle to a mond for one of those servers, and a call to get a handle to a PA that is under the control of a particular mond. When using explicit binding, a client can specify that the client block if the PA is busy or that it get back a status if the PA is busy. In addition, the client can request that the PA be reserved for that client by specifying a long-term reservation to the server.

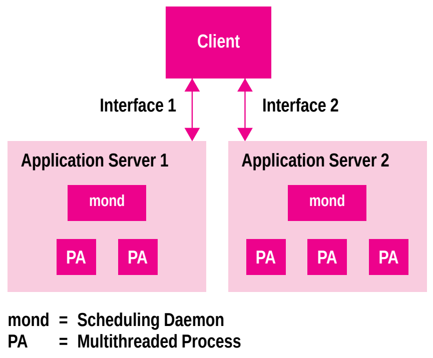


Fig. 11. An example of an encina cell's application servers.

In general it is easier to code the client to use transparent binding. This also has the advantage that the monitor code can perform load balancing of client requests among the available PAs. The monitor software uses a probabilistic algorithm to route client requests to the available PAs in a ratio predefined by a system administrator. With transparent binding the monitor software will use an existing binding if one exists, or it will create a binding to an appropriate server if no such binding exists. If all the available servers are busy, the client waits at the server for a free PA.

Although it is more complicated to write clients that use explicit binding, it does provide the user with the ability to select the PA on which the call is executed. There are certain situations in which explicit binding used in conjunction with long-term reservation of PAs is advantageous. For example, consider a client process servicing a large number of users. In this case it would be advantageous for that process to reserve a PA and then direct the various user requests to that PA. Having a direct connection to a PA reduces the time needed to connect to a PA on subsequent calls. Long-term reservation makes the PA unavailable to other clients, and it must be used with care. Administratively, a timeout interval can be specified so that if there are no client calls to the PA within that interval, the long-term reservation is canceled.

When an application server is initialized, it can specify one of the three scheduling policies: exclusive, concurrent shared, and shared. Shared scheduling is provided primarily for compatibility with previous releases, and its use is not recommended. The default policy is exclusive scheduling. With this scheduling only one client RPC can be executing within a given PA at any time, and the PA is scheduled exclusively for the entire duration of the client transaction. This has the advantage that the programmer does not have to be concerned about issues related to threading. This is required when the PA is accessing a database that is not thread safe (which is currently the case for most RDBMSs).

With concurrent shared scheduling, many clients can be executing within a PA at the same time, and the multithreaded PA assigns a different thread for each client request. If the PA accesses global or static variables, they must be protected by DCE synchronization facilities such as mutexes.* Concurrent shared scheduling should only be used when linking with thread-safe libraries. Concurrent shared scheduling provides the best performance with the lowest use of resources.

The monitor allows the creation and access of monitor-shared memory (HP 9000 virtual memory) which can be shared among the PAs within an application server. This allows a quick and easy way for the PAs to share transactional data. Monitor-shared memory is much cheaper than, say, using an external RDBMS, but care should be exercised when using the monitor-shared memory because it is the user's responsibility to perform the appropriate locking when accessing the shared memory. Since locks must be

* Mutexes, or mutual exclusion locks, are used to protect critical regions of code in DCE threads.

used, it also has the potential of introducing deadlocks. Transactional timeouts can be declared for aborting such transactions.

The monitor allows the use of the recoverable queueing service for queueing work items which are eventually processed by a monitor application server. Using the queued request facility, entries of the appropriate type are queued to the recoverable queueing service. A queued request facility daemon will then dequeue the request and forward the request to the appropriate PA.

The Encina/9000 monitor also provides a timer mechanism to allow servers to schedule a call to be issued at a later time. This functionality is provided transactionally so that the call made within the scope of a transaction is scheduled if the transaction commits. The call does not occur if the transaction aborts.

The Encina/9000 monitor provides support for application developers who wish to integrate their Encina/9000 client with forms-based user interface tools. Encina/9000 is integrated with JAM, a forms-based tool from JYACC Inc.

In summary, the Encina/9000 monitor provides the following benefits:

- Simplified programming for writing clients and servers
- Automatic detection of failures and restarts of monitor daemons and PAs
- Automatic load balancing between clients and servers
- Collection of statistics by the monitor for server use
- Simplified central place of administration for distributed clients and servers
- Support for highly concurrent access to relational databases.

Standards Supported by Encina/9000

Encina/9000 supports the following standards:

- X/Open:
  - XA
  - TX
  - TxRPC API
  - CPI-C
  - ISAM
- SAA:
  - CPIC
  - CPI/RR
- OSF DCE.

Encina/9000 interoperates with the following products:

- Oracle
- Informix
- Ingres
- Sybase
- Open CICS
- IBM mainframe CICS
- IBM mainframe IMS/DC.

The Encina/9000 toolkit has been used to support other transaction processing products and provide the base functionality to support other products like Open CICS and STDL each running on top of Encina/9000 on the HP-UX operating system.

Value-Added Features

HP Encina/9000 provides value-added features in the areas of system administration and high availability.

System Administration

An Encina/9000 system administrator must configure the Encina cell and define the administrative interfaces for the various servers in the system.

An Encina cell must be closely tied to a logical administrative unit of work, and the data accessed to do this work should be in the same cell. It is possible for applications to interoperate across Encina cells using explicit bindings. Therefore, the exact boundaries of an Encina cell must be defined by carefully analyzing the applications running in the system with careful consideration being given to security, number of users and machines, location of data, and the applications that access the data.

A system administrator must create the log space used by Encina/9000 and then bring up the following Encina/9000 components:

- Structured file system
- Cell manager
- Node manager
- Servers such as the recoverable queueing service and the PPC gateway if they are needed
- The required application servers.

Encina/9000 provides administration tools for the following components: log, structured file system, monitor, recoverable queueing service, PPC, and the rest of the toolkit. These tools provide the appropriate low-level commands for administering these components. Encina/9000 also provides a perl-based* tool called encsetup, which provides higher-level system administration facilities. The HP value-added system administration facilities are described later in this section. Finally, Encina/9000 also provides libraries for developing system administration products, which are very useful for customers developing these kinds of products.

Encina/9000 system administration is very closely tied to DCE system administration. The DCE cell must be configured before the Encina cell can be configured. Encina/9000 also makes use of the DCE directory service. The default Encina root cell directory is defined as /.:/encina (this default can be changed if needed). Encina/9000 components register their name under this directory. Within this directory there are directory entries for the recoverable queueing service, the structured file system, the Encina/9000 monitor, transactional RPC, and peer-to-peer communication (PPC). For example, each recoverable toolkit server registers an entry in the /.:/encina/trpc directory (trpc = transactional RPC), and each recoverable monitor server registers an entry in the /.:/encina/tpm/trpc directory (tpm = Encina/9000 monitor).

The use of the directory allows the Encina/9000 system administrator to restrict access to various resources. The

* Perl (Practical Extraction Report Language) is a UNIX programming language that is designed to handle system administrator functions.

system administrator can use DCE tools like acl_edit to grant a user, a group, or an organization permission to access a particular resource. Encina/9000 uses the DCE authentication and authorization mechanisms to maintain security. An Encina/9000 server can specify the level of authorization a user of the server must have to access that server. A client wishing to access a secure server must be authenticated with DCE and when the client calls the server, the server uses the DCE security mechanisms to verify whether it should allow access to the user. DCE access control and security are discussed in the articles of pages 49 and 41 respectively.

HP provides a DCAM layer for Encina/9000. DCAM stands for distributed computing application management. DCAM is an architecture and methodology for providing uniform system management for products that enable distributed computing such as DCE, Encina/9000, and CICS. An advantage of DCAM is that it provides a consistent look and feel for all of these products to the user and aids in the overall ease of use of these products. It provides a graphical user interface as well as a DCAM shell. DCAM provides a set of action verbs that can be modified by options and operate on objects.

Fig. 12 shows the relationship between the DCAM CLI (command-line interface) layer, the DCAM shell, and SAM (system administration manager).

SAM is a menu-driven interface used to manage an Encina/9000 system. The DCAM shell is a command-line interface which can be used to type in administrative commands. SAM and the DCAM shell are layered on top of the DCAM CLI scripts which convert the DCAM commands to native Encina/9000 administration commands.

The common look and feel provided by DCAM enables a system administrator to manage the different distributed systems and applications based on DCE with a consistent and user-friendly interface. DCAM does this by providing consistent use of vocabulary to represent actions. The consistent use of syntax and semantics is important because of the different subsystems that DCAM is built upon. The consistency provided by DCAM improves user efficiency and lowers error rates.

DCAM provides a natural way for system administrators to express the actions that they want. For example, to create a structured file system server, a system administrator would type the command: create sfsserver. This command is converted by DCAM to the underlying Encina/9000 low-level commands needed to create the server.

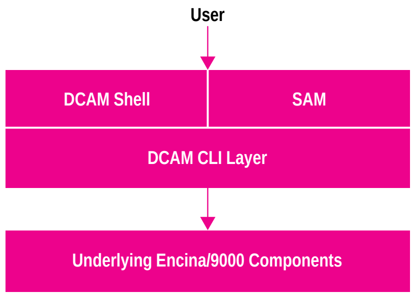


Fig. 12. System administration tools with DCAM.

The SAM interface of DCAM is more useful for people who are familiar with SAM or are getting acquainted with Encina/9000. The DCAM shell is generally used for efficiency by experienced Encina/9000 system administrators. In addition, the DCAM shell is also used for writing and customizing system administration scripts.

DCAM is object-oriented. Objects represent items that can be encapsulated and acted upon. Encina/9000 objects can be an Encina cell, a server, or a transaction. Objects have attributes. For example, a structured file system server has an associated attribute that describes the log volume associated with the server. Actions are verbs that act upon the objects. For example, the actions create, start, modify, and stop can be used to act upon an object. Actions have object independent semantics in that they have similar semantics regardless of the type of object they are working on. For example, the verb create can be used to create an Encina cell, a structured file system server, an application server, and so on. Actions have options. An action can be specified with the default options, or the administrator can specify task-specific options with the action.

A task defines a pairing of an action with an object. A task consists of one action, one object, zero or more options, and one or more attributes. For example, start cell -Name name, which tells DCAM to start up the named cell along with other optional parameters, is a task that can be specified with the DCAM shell. If the parameters are not specified, the DCAM shell will prompt for the parameters. In SAM, the parameters are displayed as fields in the SAM panel and can be entered. If the required parameters are not entered, an error is displayed.

Another useful feature of DCAM is the help facility, which can be used by the system administrator to interactively obtain help on a topic. This is also useful for someone who is learning Encina/9000 administration since it lists the various alternatives and options to a command and provides an easy way for administrators to get a feel for the various commands and options.

To many users the real value of DCAM is the added capabilities it has that go beyond what native Encina/9000 administration supports. This includes high-level server configuration tasks which are much easier, complete support for transparent remote configuration from anywhere in the DCE cell, autorestart of toolkit servers like the structured file system and the recoverable queueing service, and support for ServiceGuard/UX's failover* feature.

High-Availability Features

Many customers have a strict requirement for data to be available at all times. Data replication with Encina/9000 can be provided by the use of data mirroring with mirrored disks. In addition, to provide data availability in the case of machine failures, Encina/9000 can be integrated with the Switchover/UX and the ServiceGuard/UX products (described below). These products allow node failures to be handled, and they provide a set of scripts that facilitate the administration of a highly available system.

* Failover refers to the process that occurs when a standby node takes over from a failed node.

In a distributed system there can be many causes of failures, and failures of disks, networks, and machines can all impact availability. Since the system is composed of several nodes connected with network links, there are more points of failure that could impact availability. Network failures are not described here, but users who need highly available Encina/9000 applications should try to avoid single points of network failure.

Many techniques exist for dealing with disk failures. The preferred method of dealing with disk failures is to use HP-UX mirrored disks with a logical volume manager. Other choices are to use RAID[5] or to use Encina/9000 mirrored disks. The advantage of the HP-UX mirrored disk technique is that it is a general-purpose solution with applicability to all kinds of data like the structured file system and DBMSs. If the database can handle the logical volume manager configured for no consistency then it should be used for database data. Mirror write consistency, or mirror consistency, should be used for Encina/9000 data or for database data that can handle consistency mirroring. RAID can be used as a relatively inexpensive solution to handle disk failures, but it has many single points of failure in the disk I/O path and is not good for the short random write updates that are typically found in transaction systems. Encina/9000 mirroring has the disadvantage that it is not integrated in the HP-UX operating system and can therefore only be used for Encina/ 9000 data and not for, say, DCE or DBMS. Its advantage is that it can automatically handle more failure conditions than HP-UX mirroring. Encina/9000 mirroring is slower than HP-UX mirroring, but it has a faster recovery time.

There are two primary solutions for node failures: Switchover/UX and ServiceGuard/UX. In Switchover/UX, a primary node and a standby node are configured with multihosted disks. The primary node runs in the normal case. The standby node is also connected to the disks and uses a heartbeat protocol to detect failure of the primary node. When the standby node detects that the primary node has failed, it assumes the primary node's identity by booting off the primary node's disks and using the primary node's IP address. The standby node then uses the primary node's disks to reboot and to restart the system processes and applications. This allows a fast restart after the primary node has crashed, resulting in a small downtime. The primary and standby nodes should be from the same hardware family.

With Switchover/UX the Encina/9000 processes are restarted when the standby node reboots. Using Encina/9000's transparent binding, clients are automatically reconnected to the servers. However, in this case client transactions will keep aborting until the failover is complete.

In ServiceGuard/UX, applications and data are encapsulated as packages that can be run on various nodes of a cluster. ServiceGuard/UX allows the user to define the packages, and each package has a prioritized list of nodes it can run on. ServiceGuard/UX ensures that a package only runs on one node at a time. A package is defined by a startup/shutdown script and can represent any application. The nodes running packages monitor each other's status and restart packages when they detect the failure of another node.

A package can be an Encina/9000 application server running under a single Encina/9000 node manager. The package can also include assorted toolkit servers like the structured file system, a recoverable queueing service, or an Encina/9000 monitor. Optionally, a package can have one or more IP addresses. If specified, a package's IP address is associated with the network interface on the machine currently executing the package. With ServiceGuard/UX a user can configure a simple failover scheme. The user can also define a single package that can execute on a primary or a backup node. This scheme is general and can be used for the Encina/9000 log, structured file system, recoverable queueing service, monitor, and DBMS and DCE core servers.

Encina/9000 servers can be integrated with ServiceGuard/UX. In this case the Encina/9000 servers should be configured in a ServiceGuard/UX cluster, and a package should be created for the servers. The package should contain run and halt scripts for the servers, which specify the actions to take when a package is started or terminated. The actions in a run script include adding the relocatable IP address to the network interface, mounting all logical volumes, and calling an Encina/9000 script to start all the Encina/9000 servers. The actions in a halt script include calling an Encina/9000 script to halt all the Encina/9000 servers, unmounting all the logical volumes, and removing the relocatable IP address.

ServiceGuard/UX offers a more flexible solution for high availability. It can be configured with a dedicated standby solution similar to Switchover/UX, or it can be configured in a more cluster-like configuration. It also has a faster recovery time since failover nodes do not need to reboot.

Encina/9000 also provides the ability to perform application-level replication of data. An alternative to application-level replication is the replication of data provided by databases. Database-level replication has the advantage of being transparent to the user, and it is relatively efficient. Application-level replication, on the other hand, is less dependent on specific DBMS platforms and can be used to provide replication across DBMS platforms. In addition, it is more flexible and can be performed in a synchronous or asynchronous manner. It may be important to perform asynchronous replication across a WAN to achieve a faster response time. The disadvantage of application-level replication is that the application developer must design and implement the replication scheme.

An example of replication using Encina/9000 is master/slave replication of data with deferred updates to the slave. In this scheme, the master copy of the database is maintained on a machine. The application updates the master database and stores a copy of the update in the recoverable queueing service. With this setup the application can transactionally update the master database and store a copy of the updates in the recoverable queueing service. At a later time the data is transactionally dequeued from the recoverable queueing service and applied to the slave database on another machine. The strength of this approach is that the two machines holding copies of the data do not have to be running at the same time, and the update can be deferred to a time when the load on the system is low. It also avoids having to do online a two-phase commit across the machines. However, there is the drawback that the replica is not consistent with the master, and the updated data would be unavailable while the master node is down.

Encina/9000-Based Architectures

Some of the common Encina/9000-based architectures include corporate centralized data architecture, region centralized data architecture, and branch data architecture. In each of these architectures we consider a corporation to be an entity that has a central data processing center located at its headquarters. The corporation's business is geographically spread over several regions, and each regional center has a data processing center. Each region also contains multiple branch offices, and the branch office has a number of users who are executing transactions. In the past, companies employed mainframes at the corporate headquarters, where all the data was maintained. This was expensive to maintain, and the response time got worse as the data on the mainframe increased.

In a corporate centralized data architecture, data is still maintained at the mainframe host. Connection to the mainframe is through gateway machines that run the Encina/9000 PPC executive. Depending on the availability requirements, the gateway machines could be implemented with the high-availability solutions mentioned earlier. One option would be not to have any machines at the branch offices or the region offices but rather to have PCs at these offices which talk directly to the mainframe. Alternately, the regional centers or the branches could have machines, and the regional machine could route a request from a branch to the corporate center. All the data is maintained at the corporate center and there is no local business data at either the regional offices or the branch offices. This architecture is shown in Fig. 13.

This architecture is useful if it is hard to replace the mainframe machines and data. Alternately, it may be possible to offload some of the data from the mainframe to machines running the HP-UX operating system at the corporate data center.

The regional centralized data architecture is similar to the corporate centralized data architecture, except that the data is partitioned across the regional data centers. The data could also be stored with the mainframe at the corporate data center. Clients can run at the branch machines or at the regional center. Optionally, there could be a database at either the corporate center or the regions that assists in routing a request to the appropriate regional center. This architecture is shown in Fig. 14.

In the regional centralized data architecture, Encina/9000 servers typically run on the regional machines. Clients can run at the branch or regional offices. Clients employ lookup mechanisms to locate the appropriate server and then make calls to the servers. An Encina/9000 PPC can be used to transactionally read or update data stored at the corporate center.

The regional centralized data architecture has the advantage of avoiding CPU bottleneck problems when a large number of transactions have to be processed on a single database. Since the databases are spread throughout the regions, they all can handle transactional access to the data allowing a higher volume of transactional traffic. In addition, if users frequently access data at the nearby regional center, network traffic will be localized.

In the branch data architecture, each branch maintains its local branch data. The data can also be aggregated and maintained at the corporate center, but users primarily access the data at a branch machine. Optionally, corporate or regional centers can maintain a cross-reference database to assist with routing a user request to the appropriate branch. This architecture is shown in Fig. 15.

The main advantage of this solution is the fast response time, since for most transactions data can be looked up locally and expensive two-phase commits over the WAN can be minimized. The drawback of this scheme is having to maintain a large number of databases and administering them.

Conclusion

The Encina/9000 product provides an application development environment for developing OLTP applications and the run-time support for running and administering the applications. Its strengths are the flexibility it provides for distributed OLTP applications compared to the traditional database products, and its strong integration with the HP DCE product. It provides an infrastructure for customers to write reliable

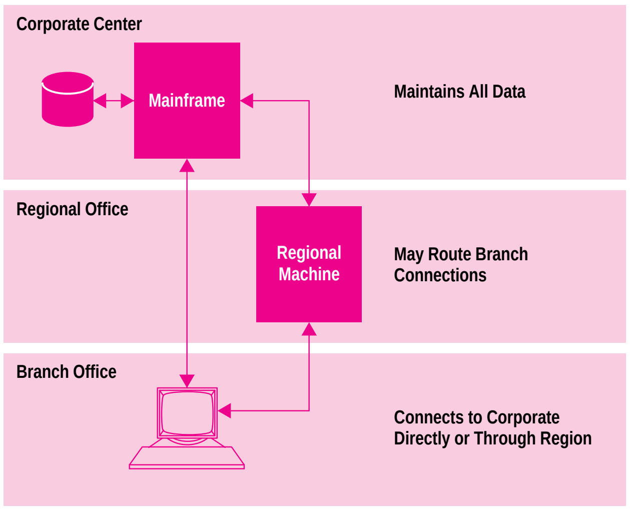


Fig. 13. A corporate centralized data architecture.

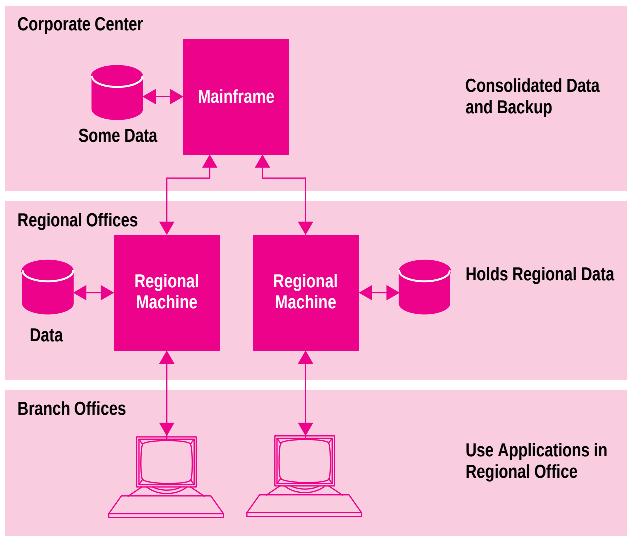


Fig. 14. A regional centralized data architecture.

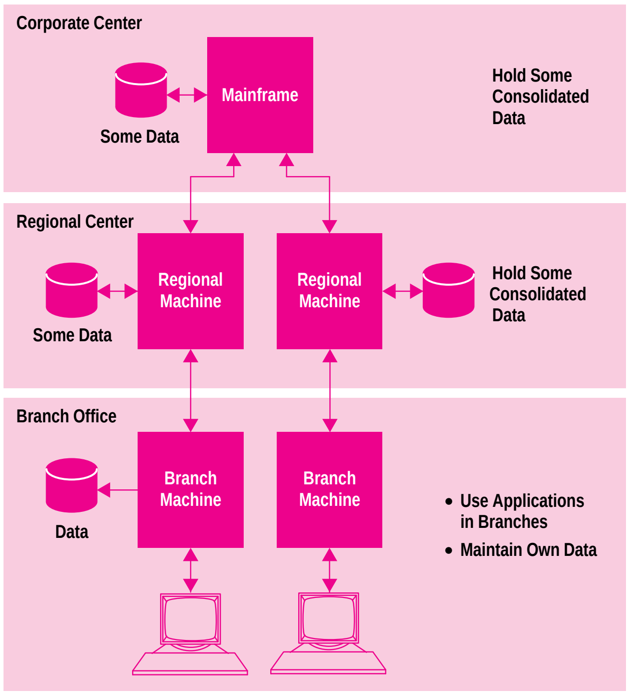


Fig. 15. Branch data architecture.

and secure applications for their mission-critical data. Additionally, the Encina/9000 product provides added value in the areas of system administration and fault tolerance.

Acknowledgments

I would like to thank Jay Kasi for his insightful discussions on several topics mentioned in this paper.

References:

1. *Encina/9000 Reference Manuals,* Part Number B 3789AA, Hewlett Packard Company, 1995.

2. *OSF DCE Application Development Guide*, Revision 1.03, Prentice Hall, 1993.

3. J.E.B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing,* MIT Press, 1985.

4. *CAE Specification Distributed Transaction Processing: The XA Specification,* X/Open, 1991.

5. M. Rusnack and T. Skeie, HP Disk Array: Mass Storage Fault Tolerance for PC Servers, *Hewlett-Packard Journal,* Vol. 46, no. 3, June 1995, pp. 71 to 81.

Object-Oriented Perspective on Software System Testing in a Distributed Environment

A flexible object-oriented test system was developed to deal with the testing challenges imposed by software systems that run in distributed client/server environments.

by Mark C. Campbell, David K. Hinds, Ana V. Kapetanakis, David S. Levin, Stephen J. McFarland, David J. Miller, and J. Scott Southworth

In recent years software technology has evolved from single-machine applications to multimachine applications (the realm of the client and server). Also, object-oriented programming techniques have been gaining ground on procedural programming languages and practices. Recently, test engineers have focused on techniques for testing objects. However, the design and implementation of the test tools and code have remained largely procedural in nature.

This paper will describe the object testing framework, which is a software testing framework designed to meet the testing needs of new software technologies and take advantage of object-oriented techniques to maximize flexibility and tool reuse.

System Software Testing

The levels of software testing include unit, integration, and system testing. Unit testing involves testing individual system modules by themselves, integration testing involves testing the individual modules working together, and system testing involves testing the whole product in its actual or simulated operating environment. This paper focuses on software system testing.

A software system test is intended to determine whether the software product is ready to ship by observing how the product performs over time while attempting to simulate its real use. System testing is composed of functional, performance, and stress tests. It also covers operational, installation, and usability aspects of the product and may include destructive and concurrence testing. The product may support many different hardware and software configurations which all require testing. All of these aspects are combined to assess the product's overall reliability. Software system testing is usually done when all of the individual software product components are completed and assembled into the final product.

In the past, system testing environments centered around testing procedural, nondistributed software. These environments, which were also procedural and nondistributed, were usually developed by the test writer on an ad hoc basis along with the test code for the product. Recently, software system testing has benefited from the use of highly automated test harnesses and environments that simplify test execution, results gathering, and report generation (see Fig. 1). Unfortunately, the test harnesses created in these environments were not easily reusable, and when the next project reached the test planning stage, the test harness had to be reworked.

The advent of standardized test environments such as TET (Test Environment Toolkit)* helped to reduce this costly retooling by providing a standard API (application program interface) and tool base that test developers can adopt and use to write standardized tests. However, the difficulty is to provide a standard test harness that is complete but flexible enough to keep pace with changing software technology and remain viable for the long term.

During the development and testing of the initial release of HP ORB Plus, which is an object request broker based on the Object Management Group's CORBA specification (see page 76), we realized that distributed object technology posed testing problems that were not adequately solved by any of the test harnesses currently available. We needed a flexible test environment that could handle heterogeneous

* The Test Environment Toolkit (TET) specification began in September 1989 as a joint proposal by the Open Software Foundation, UNIX® International, and X/Open®.

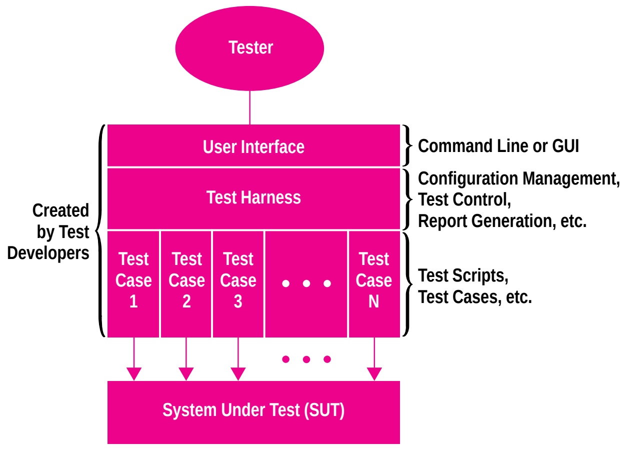


Fig. 1. A typical automated test environment.

distributed systems communicating over multiple transports using multithreaded clients and servers. However, we were not willing to lose the investment we made in the test code and tools developed for our earlier products.

Instead of abandoning the old test environment and replacing it with an entirely new system, we decided to use the object-oriented principles of encapsulation and polymorphism to evolve our current environment base to meet our needs without throwing out the old code. The ability to change or replace functional blocks of a system without affecting the entire environment is one of the main benefits of object-oriented design (see "Object-Oriented Programming" on page 79). Object-oriented principles allowed us to reuse existing tools.

Distributed System Testing

In a distributed object system, service providers are distributed across a network of systems and are called servers. Clients, who issue requests for those services, are also distributed across a network. A single program can be both a client (issues requests) and a server (services requests). Clients can issue requests to local and remote servers. During a distributed object system test, clients are responsible for reporting any failures or status resulting from the requests they make.

The first task performed during the system testing of a distributed object software product is test setup. Clients and servers must be deployed across the network to targeted systems. Consideration must also be given to the fact that servers may have multiple clients sending messages to them, and the distribution of clients and servers may change during a system test so that various hardware and software configurations can be tested.

The Object Management Group's Distributed Object Model

The Object Management Group (OMG) creates standards for the interoperability and portability of distributed object-oriented applications. The OMG only produces specifications, not software. Each participating vendor provides an implementation to meet the specification. The Common Object Request Broker Architecture (CORBA) specification defines a flexible framework upon which distributed object-oriented applications can be built. This architecture represents the Object Request Broker (ORB) technology that was adopted by the OMG. An ORB provides the mechanisms by which distributed objects transparently make requests and receive responses. The ORB enables object-oriented applications on different machines to communicate and interoperate.

The OMG has defined an Object Management Architecture object model. In this model, objects provide services, and clients issue requests for those services. The ORB facilitates this model by delivering requests to objects and returning any output values to the client. The services that the ORB provides are transparent to the client.

To request a service, a client needs the object reference for the object that is to provide the service. The ORB uses this object reference to identify and locate the object. The ORB activates the object if it is not already executing and directs the request to the object.

When test clients execute, they are instructed to run for a specified amount of time. They report failure and status information back to a central location. Upon completion of the system test, clients and servers are stopped, temporary files are removed, and final summary reports are produced.

The Test System

To manage all the activities of distributed system testing, we developed a test infrastructure that met our current needs and could evolve with new technologies and new needs. We followed a modular, object-oriented design approach to accomplish this.

We first engaged in several brainstorming sessions to produce a list of requirements for a complete distributed testing framework. This was an attempt to pinpoint all the attributes and functionality that a "perfect" test infrastructure would have, and it was done in the context of system testing distributed objects. The needs of product developers, test developers, and testers were considered, as well as the need to report metrics to the project team. The main focus, however, was on the two groups who would use the test framework the most: test developers and testers. Often these are the same people, but the distinction was made to clearly differentiate the needs of each group.

Product developers normally want quick and simple tests to verify that their code behaves correctly and at the same time have their programs work as they would for an end user. They don't want to be distracted by the infrastructure. Existing test APIs tend to be intrusive, requiring developers to have knowledge of the test environment in which their tests will be run. Therefore, we wanted our new test framework to minimize intrusiveness. This would allow developers to focus on testing the proper behavior of their code and not on the test infrastructure. Ideally, product developers should be able to write their tests with minimum restrictions, and the tests should plug and play in many different testing situations.

Test developers, whose job it is to develop ways to test the product, have many of the same needs as product developers but are more concerned with black-box testing and trying to "break" the product rather than verifying correct behavior. To do this, test developers want to be able to plug new tests into the test environment easily and quickly, and they want process and environment control. This would allow them to use the same tests in different scenarios to find more product defects. Test developers are usually the ones responsible for supporting the testing infrastructure. Thus, more than any of the other groups, test developers need a framework that is extensible, reusable, flexible, and controlled, and hopefully has a long lifetime. If a testing infrastructure becomes out of date, test developers will have to repair or replace it.

Often test developers are the ones who perform system testing, but many times this role is handed off to testers. Although the needs of both groups clearly overlap, testers need a testing infrastructure that is easy to use for the installation, configuration, and execution of tests. In many of our past projects, testing was done by temporary personnel. This freed test developers to write more tests and assist product developers in debugging. When the test infrastructure is easy to use, the testing role can be handed off to testers earlier in the testing process. Additionally, the ability to reconfigure the test environment easily and quickly allows more scenarios to be tested. This increases the likelihood of finding more product defects, which leads to a better quality product.

Finally, test results are usually provided to the project team in the form of metrics. Gathering metrics in a distributed environment can be time-consuming. Data can be located on multiple systems on the network. However, when dealing with multiple processes running in parallel on different systems, results may not always occur in a consistent order. This implies the need for a centralized repository for testing results. This would make the generation of metrics much easier and faster, while providing a central location for finding problems and debugging.

Design Methodology

Taking into consideration the needs of the different groups mentioned above, we decided that the following attributes were required for our test infrastructure.

- Extensibility. Ensure the evolution of a modular system that can be dealt with on a component-by-component basis.
- Reusability. Allow object and code reuse for both tests and the test infrastructure.
- Flexibility. Provide a plug-and-play environment that allows for flexibility in test writing and configuration.
- Simulation. Provide the ability to simulate customer environments.
- Control. Provide centralized control of the test processes and environment.
- Nonintrusive. Hide as much of the testing infrastructure as possible from the system under test.
- Ease of use. Provide ease of use for installation, setup, configuration, execution, results gathering , and test distribution.

With these attributes in mind, we set about deciding on the basic set of classes that would be needed. We used a method for object-oriented design called Object Modeling Technique (OMT)[1] to develop a diagram showing class relationships (see “Object-Oriented Programming” on page 79).

We walked through several scenarios and expanded and refined our set of classes. Once we had an initial design we wrote CRC (class, responsibility, and collaboration)[2] cards for each of the classes in our design. (CRC cards are also described on page 79.) This design was reviewed by the product development team and their feedback was incorporated.

The Object Testing Framework

The design process produced an object-oriented software testing system that we named the object testing framework (OTF). Although this design is intended to test distributed object-based software, it can also be used to test distributed, procedurally based client/server software. The OTF consists of the classes shown in Fig. 2. The architecture of the OTF is such that there is a single master test control system (OTF management system in Fig. 2) that orchestrates running tests on multiple systems under test. This master system can also be a system under test.

In the following design discussion, the term object can mean class or an instance of a class. It should be clear from the context of the discussion which is meant.

OTF Management System

The OTF management system consists of the six major classes: user interface, OTF controller, test suite configuration, test controller, report generator, and database controller. This system provides the user interface that the software tester interacts with. Through this interface the tester specifies test configurations such as which client and server programs will be running on which SUTs. The OTF management system takes the specified configurations and makes them available to each of the SUTs, ensures that the SUTs run the specified tests, logs test data and results, and generates test reports.

The main class in the OTF management system is the OTF controller, which serves as the delegator object. It takes requests from the user interface object and manages the activities of the test suite configuration, test controller, and report generator objects. The test suite configuration object is actually created by the OTF controller. For a new configuration the object will initialize from the configuration data provided by the user interface. For a previously specified configuration, the object will initialize from the database. After this object’s configuration data has been set, its primary responsibility is to respond to configuration queries from the SUTs.

The test controller has the overall responsibility for coordinating the running of tests on the SUTs. It provides the SUTs with a pointer to the test suite configuration object, synchronizes the starting of tests, and passes status data and requests back to the OTF controller. It also has the capability to log status data to the database via the database controller.

The report generator, upon a request from the OTF controller, queries the database controller to assemble, filter, and format test data into user-specified test reports. Raw test data is put into the database by each SUT’s TestEnvironment object, while test process status data is put into the database by the test controller as mentioned above.

System under Test

Each system under test (SUT) contains fifteen classes. In normal operation, a SUT retrieves configuration data from the OTF management system, and then, based on that data, retrieves the specified tests from the management system. Since the SUT has the capability to build test executables from source code, it can retrieve test source code and executables from the OTF management system. Once the test executables are in place and any specified test setup has been completed, the SUT waits for a management system request to start the tests. When this happens, the SUT is responsible for running the tests, logging status, test data, and results, and cleaning up upon test completion.

The main object in the SUT is the host daemon, which is the SUT’s delegator object. The host daemon takes requests from and forwards requests to the OTF management system and manages the activities of the setter upper, test executor, cleaner upper, process controller, and TestEnvironment objects.

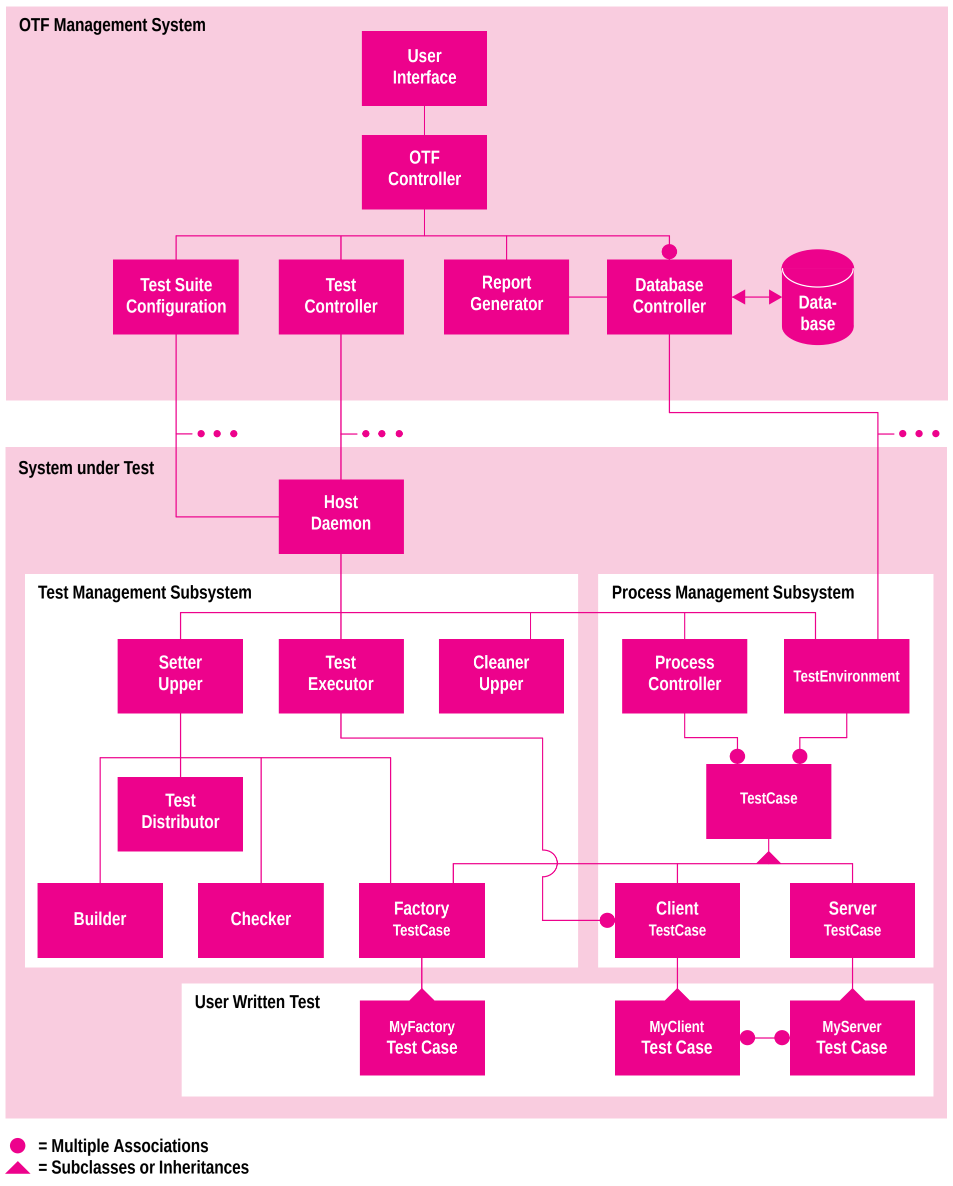


Fig. 2. System architecture for the object testing framework.

The overall responsibility of the setter upper, test executor, and cleaner upper objects is to manage how the tests are run. These three objects collaborate with the builder, test distributor, checker, and factory TestCase objects to form the test management subsystem shown in Fig. 2. The process controller and TestEnvironment objects provide the infrastructure for connecting the tests to the framework. These two objects collaborate with the TestCase objects to form the process management subsystem.

Test Management Subsystem

This subsystem sets up and executes the tests and then cleans up after the tests have completed. The setter upper is the object that controls test setup. It is a low-level delegator that manages the activities of the builder, test distributor, checker, and factory TestCase objects. The test distributor is responsible for retrieving test executables and sources from the OTF management system. When it retrieves source code, the builder is responsible for generating test executables from the code. How the tests are retrieved depends on the overall system environment and resources available. A distributed file system, like NFS, could be used, or the tests could be remote copied from the management system to the SUT. An important design consideration was to have a single repository for tests. This makes it easy to control changes to tests and is not intrusive on the SUTs.

The checker provides the ability to customize test setup by invoking a user-written program that can ensure that elements outside of the test environment are set up correctly. For example, it could check that NFS and DCE are running, that the display is set correctly, and so on.

The factory TestCase provides the setup procedures that arise when testing a CORBA-based distributed object system. It creates the CORBA objects that reside in the CORBA-based server TestCases and stores references to these objects for use by the client TestCases. The factory TestCase class inherits from the TestCase base class and the test developer writes a class that inherits from the factory TestCase class. This allows the test developer to customize the factory TestCase functionality for a specific test.

The test executor object starts the client TestCases through the functionality inherited from the TestCase base class. It also

Object-Oriented Programming

Object-oriented programming is a set of techniques for creating objects and assembling them into a system that provides a desired functionality. An object is a software model composed of data and operations. An object's operations are used to manipulate its data. Objects may model things such as queues, stacks, windows, or circuit components. When assembled into a system, the objects function as delegators, devices, or models. Messages are passed between the objects to produce the desired results.

The eventual goal of object-oriented programming is to reduce the cost of software development by creating reusable and maintainable code. This is achieved by using three features of object-oriented programming: encapsulation, polymorphism, and inheritance. Encapsulation consists of data abstraction and data hiding. Data abstraction is a process of isolating attributes or qualities of something into an object model. With data hiding an object may contain its own private data and methods, which it uses to perform its operations. By using polymorphism, an object's operation may respond to many different types of data (e.g., graphical and textual). Finally, using inheritance, an object can inherit attributes from other objects. The object may then only need to add the attributes, methods, and data structures that make it different from the object from which it inherited its basic characteristics.

For the design of the object testing framework described in the accompanying article, we used an object-oriented software design methodology called object modeling technique (OMT). This methodology provides a collection of techniques and notation for designing an object-oriented application.

One important aspect of object-oriented design, or any software design, is deciding on who (i.e., module or object) is responsible for doing what. A technique provided in OMT involves using an index card to represent object classes. These cards are called CRC (class, responsibility, and collaboration) cards. The information on one of these cards includes the name of the class being described, a description of the problem the class is supposed to solve, and a list of other classes that provide services needed by the class being defined.

reports back to the host daemon the success or failure of a test start.

The cleaner upper cleans up after the tests have completed. This may include removing temporary files, removing test executables, and so on.

Process Management Subsystem

The two main objects in the process management subsystem are the process controller and TestEnvironment objects. The process controller has the overall responsibility to monitor all test-related processes on the SUT. It can register or unregister processes, kill processes, and report process status back to the host daemon.

The TestEnvironment class provides the test developer with an application programming interface to the OTF. It provides methods for aborting tests, logging test data and results, checking for exceptions, getting environment variables, and so on. The test developer gets access to these methods through the base TestCase class, which has an association with the TestEnvironment class.

Creating a test involves writing a class that inherits from either the client TestCase or server TestCase base classes. The initialization and setup functionality for the test would be included in the test's constructor. The cleanup required when the test is done is included in the destructor. Finally, an implementation for the inherited run_body() method is included, which is the test executable that runs the test. The OTF API is made available through the pointer to the TestEnvironment class provided by the base TestCase class.

Implementation Approach

Once the design was complete, an initial investigation was made to find an existing system that matched the characteristics of the design. When no system was found, an analysis was done to determine the cost of implementing the new infrastructure.

It quickly became obvious that the transition to the new infrastructure would have to be gradual since we did not want to impact the HP ORB Plus product release cycle. The flexibility provided by an object-oriented system enables gradual migration and evolution through encapsulation, inheritance, and polymorphism. Tests could be isolated from the infrastructure so that new tests could be developed and evolved without modification as the infrastructure evolved. This flexibility fit nicely with the realization that the time to replace the existing infrastructure exceeded an average product life cycle.

Object-oriented encapsulation provided another advantage. Once some basic changes were made to the existing test infrastructure and tests had been converted to the new object-oriented programming model, the existing test infrastructure could be used to simulate some aspects of the new infrastructure. This allowed our system testing efforts to benefit immediately from the features of the new test system.

The development of the current version of the object testing framework has taken place in two steps, which have spanned three releases of the HP ORB Plus product. At each step we have continued to apply the same design principles. This work is summarized in the following sections.

First Step. For the first step, the goal was to consolidate the best practices of three existing test infrastructures into a single infrastructure that simulated as much of the major functionality of the OTF as possible. So as not to impact the ongoing HP ORB Plus software releases, another goal was to minimize changes to existing test code. This resulted in an infrastructure that consisted of a layer of shell scripts on top of two existing test harness tools. This significantly reduced the effort needed to set up, administer, and update the network of systems that were used to system test the HP ORB Plus product, while the tests continued to use existing APIs. It also confirmed that our design was indeed trying to solve the right problems.

Second Step. For this step the goal was to deploy the test developer's API to the OTF. The result was the implementation of the C++ TestEnvironment and TestCase classes described above.

Additional classes were designed to connect the TestEnvironment and TestCase classes to the existing infrastructure, but their existence is hidden from the test developer. This provides a stable API without limiting future enhancements to the infrastructure. Once the new infrastructure was deployed, we focused on porting existing tests to it.

This framework has resulted in minimal changes to existing tests and maximum increase in functionality for the tests. Most of the work simply involved taking existing code and wrapping it in the appropriate class. All of our tests have benefited from the features provided by the TestEnvironment and TestCase classes and are insulated from changes to the framework.

At this stage of its development the object testing framework allowed the removal of intrusive test code required by the old test APIs. For example, many tests included code that allowed a test to be reexecuted for a specific time period. That code was removed from the tests because the same functionality is now provided by the framework.

In addition to supporting the design shown in Fig. 2, our current implementation provides the following functionality:

- Iteration. A test can be executed repetitively, either by specifying a number of iterations or the amount of time.
- Context-sensitive execution. The object testing framework behaves differently depending on how it is invoked. In the developer's environment it is transparent and does not affect test behavior. In the testing environment it is bound to the test system. For example, in the developer environment, the C++ functions cerr and cout go to the terminal, but in the testing environment they go to a file and the test report journal respectively. This encourages developers to put all existing tests into the framework because the test continues to work the way it did before it was ported.
- Simple naming service. A naming service allows the user to associate a symbolic name with a particular value such as the path name of a data file. In a distributed system, it is necessary for multiple processes to share values that are obtained outside of the system—for example, object references.
- Automatic capture of standard output streams cout and cerr. To simplify porting of existing tests, the cout and cerr streams are mapped to a file and the journal file respectively.
- Encapsulation of functions from the product under test. For example, the parts of CORBA used by the tests are encapsulated. As CORBA evolves and changes its C++ language bindings, only a single copy of the bindings in the framework has to change.
- Inheritance and reuse. Inheritance allows the test case developer to describe similar tests as a family of test cases derived from a common class (which in turn is derived from the TestCase class). In this case, polymorphism allows test code to be reused in multiple tests, while allowing changes to specific operations and data when needed.

Example. Our experience with the current framework has shown that the time to port existing applications tests to the new API is minimal. Fig. 3 shows an example of how test code would look before and after being ported to the new test infrastructure. This example is an implementation of the client for an OMG CORBA program, which simply prints "hello, world." In this case, the phrase will be printed by the say_it method provided by the server code. The following descriptions point out some of the differences between the two source files. The numbers associated with the descriptions correspond to the numbers in Fig. 3.

1. Fig. 3b shows portions of the test code with TestCase and TestEnvironment instrumentation. These classes are not in the code in Fig. 3a.

2. Fig. 3b includes the class declaration and method definitions of a HelloWorldTest class and a macro to register the definition with the TestEnvironment. The HelloWorldTest class is derived from the TestCase base class. Fig. 3a source has no HelloWorldTest class.

3. The check_ev_and_ptr macro in the Fig. 3a source is greatly simplified in the Fig. 3b source, thanks to the TestEnvironment's print_exception and is_nil methods.

4. Fig. 3a has a main function, whereas in Fig. 3b, the main function is replaced by the HelloWorldTest constructor, destructor, and run_body methods. This structure allows the OTF to instantiate and run the test code as needed. The constructor and destructor allow the test writer to separate out "execute once" code if desired. The run_body method may be executed more than once.

5. Fig. 3a uses a file to store the object reference string created by the server. (Use of files is potentially difficult if the client and server are on different machines.) Fig. 3b uses the TestEnvironment's naming service to get the object reference string.

6. In Fig. 3b argc and argv are not available as input parameters and must be obtained from the associated TestEnvironment.

7. The int return parameter of the main function in Fig. 3a is replaced by the TestEnvironment::Result of the run_body method in Fig. 3b. The effect is the same, to return the success or failure of the invocation.

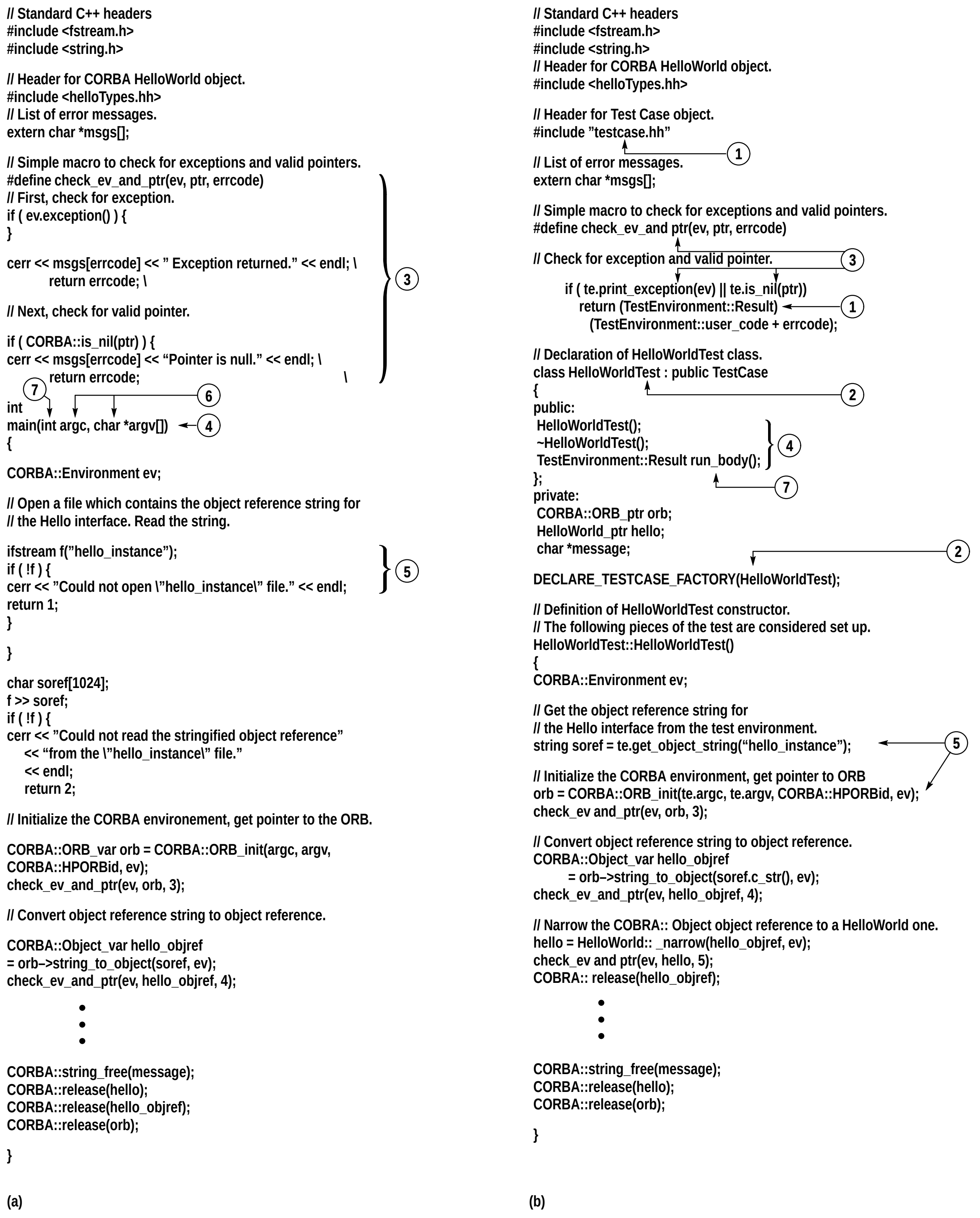
Next Steps. The following is a list of items under consideration for implementing the rest of the design and adding more functionality to the TestEnvironment and TestCase classes.

- User interface class. We are investigating the possibility of encapsulating a graphical user interface that was designed for one of the existing test infrastructures.
- Test controller class. Here again we are looking at encapsulating an existing test synchronization controller.
- Memory leak detection. By adding this feature to the TestEnvironment and TestCase classes, all tests will get this functionality through inheritance.
- Integration with run-time debugging. This will improve tracing and fault isolation in a distributed, multithreaded environment.
- Heterogeneous networks. The current object testing framework handles networks of HP-UX* systems only. We need to expand the framework to handle other UNIX® systems as well as PC operating systems.

Summary

The object testing framework is based on using object-oriented technology to create a test infrastructure that is based on a number of small, self-contained modules and then developing these modules in a way that allows the testing effort to proceed while the test infrastructure continues to evolve. Each step of the evolution results in a usable test infrastructure that keeps the test effort online and provides critically needed support to product releases.

In addition to our overall commitment to complete this project, and a desire to see it used in other organizations in HP



```
// Standard C++ headers
#include <fstream.h>
#include <string.h>

// Header for CORBA HelloWorld object.
#include <helloTypes.hh>
// List of error messages.
extern char *msgs[];

// Simple macro to check for exceptions and valid pointers.
#define check_ev_and_ptr(ev, ptr, errcode)
// First, check for exception.
if ( ev.exception() ) {
}

cerr << msgs[errcode] << " Exception returned." << endl; \
        return errcode; \

// Next, check for valid pointer.

if ( CORBA::is_nil(ptr) ) {
cerr << msgs[errcode] << "Pointer is null." << endl; \
        return errcode;                                  \

int
main(int argc, char *argv[])
{

CORBA::Environment ev;

// Open a file which contains the object reference string for
// the Hello interface. Read the string.

ifstream f("hello_instance");
if ( !f ) {
cerr << "Could not open \"hello_instance\" file." << endl;
return 1;
}

}

char soref[1024];
f >> soref;
if ( !f ) {
cerr << "Could not read the stringified object reference"
    << "from the \"hello_instance\" file."
    << endl;
    return 2;

// Initialize the CORBA environement, get pointer to the ORB.

CORBA::ORB_var orb = CORBA::ORB_init(argc, argv,
CORBA::HPORBid, ev);
check_ev_and_ptr(ev, orb, 3);

// Convert object reference string to object reference.

CORBA::Object_var hello_objref
= orb–>string_to_object(soref, ev);
check_ev_and_ptr(ev, hello_objref, 4);

        •
        •
        •

CORBA::string_free(message);
CORBA::release(hello);
CORBA::release(hello_objref);
CORBA::release(orb);

}
```

(a)

```
// Standard C++ headers
#include <fstream.h>
#include <string.h>
// Header for CORBA HelloWorld object.
#include <helloTypes.hh>

// Header for Test Case object.
#include "testcase.hh"

// List of error messages.
extern char *msgs[];

// Simple macro to check for exceptions and valid pointers.
#define check_ev_and ptr(ev, ptr, errcode)

// Check for exception and valid pointer.

     if ( te.print_exception(ev) || te.is_nil(ptr))
       return (TestEnvironment::Result)
         (TestEnvironment::user_code + errcode);

// Declaration of HelloWorldTest class.
class HelloWorldTest : public TestCase
{
public:
 HelloWorldTest();
 ~HelloWorldTest();
 TestEnvironment::Result run_body();
};
private:
 CORBA::ORB_ptr orb;
 HelloWorld_ptr hello;
 char *message;

DECLARE_TESTCASE_FACTORY(HelloWorldTest);

// Definition of HelloWorldTest constructor.
// The following pieces of the test are considered set up.
HelloWorldTest::HelloWorldTest()
{
CORBA::Environment ev;

// Get the object reference string for
// the Hello interface from the test environment.
string soref = te.get_object_string("hello_instance");

// Initialize the CORBA environment, get pointer to ORB
orb = CORBA::ORB_init(te.argc, te.argv, CORBA::HPORBid, ev);
check_ev and_ptr(ev, orb, 3);

// Convert object reference string to object reference.
CORBA::Object_var hello_objref
        = orb–>string_to_object(soref.c_str(), ev);
check_ev_and_ptr(ev, hello_objref, 4);

// Narrow the COBRA:: Object object reference to a HelloWorld one.
hello = HelloWorld:: _narrow(hello_objref, ev);
check_ev and ptr(ev, hello, 5);
COBRA:: release(hello_objref);

        •
        •
        •

CORBA::string_free(message);
CORBA::release(hello);
CORBA::release(orb);

}
```

(b)

Fig. 3. (a) Test code before being ported to the object testing framework. (b) The same test after being ported.

involved in distributed object technology, we will continue to participate in standards organizations such as OMG and X/Open to follow the work that is being done in the area of testing. To date, we have evaluated and provided feedback to the X/Open Consortium and have a representative monitoring the activity at OMG.

References

1. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

2. K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," *SIGPLAN Notices,* Vol. 24, no. 10, October 1989.

A New, Lightweight Fetal Telemetry System

The HP Series 50 T fetal telemetry system combines both external and internal monitoring of the fetus in a small, lightweight transmitter that is easy and comfortable for the patient to carry. It is useful for monitoring in labor, monitoring of high-risk patients, monitoring in transit, antepartum nonstress testing, and monitoring in the bath.

by Andreas Boos, Michelle Houghton Jagger, Günter W. Paret, and Jürgen W. Hausmann

Electronic fetal monitoring records fetal heart rate, uterine activity, and fetal movements onto a trace, allowing obstetrical clinicians to better assess fetal well-being and the adequacy of fetal oxygenation.

In today's high-tech hospital environment it is easy to overlook the fact that the majority of pregnant women who are admitted to the hospital to give birth are not sick, but are experiencing a natural event, the delivery of their babies. With this in mind, many hospitals worldwide are anxious to create a more friendly environment in their labor and delivery departments by reducing the amount of technology at the patient's bedside. This reduction in technology can present a problem. Although patients want a more natural environment, the nursing staff still wants to be able to oversee fetal well-being during labor and delivery. There has to be a balance between these two goals, and monitoring of the fetus via telemetry offers a solution.

Telemetry monitoring of the fetus involves connecting a patient to a radio frequency transmitter, which she is able to carry (Fig. 1). This transmits the fetal information via UHF radio frequencies to a receiver connected to a fetal monitor. The monitor records the information as if the patient were connected directly to it. The fetal monitor and receiver can be placed in a central location for the nursing staff to view the fetal information, and need not be in the patient's room, thereby reducing the perception of technology at her bedside.

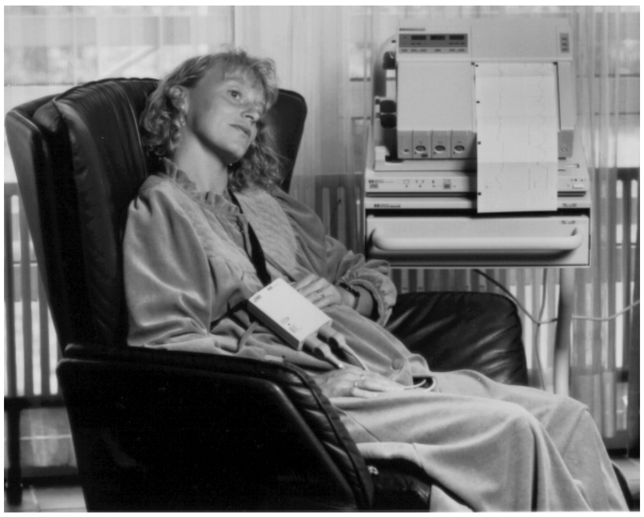


Fig. 1. The HP Series 50 T fetal telemetry system transmitter is lightweight and comfortable to wear.

Fetal monitoring with telemetry has been available for the past ten years. Until now, these telemetry systems only allowed either external monitoring of the fetus such as ultrasound detection of the fetal heart rate, or internal methods such as direct monitoring of the fetal heart rate by means of a scalp electrode. Very few systems offered both of these methods, and those that did were large and heavy for the patient to carry, and had a very low battery life.

The Hewlett-Packard Series 50 T fetal telemetry system (HP M1310A) is a new lightweight, space-saving telemetry system. It combines both external and internal monitoring of the fetus in a small, lightweight transmitter that is easy and comfortable for the patient to carry. Because the patient is not connected directly to the fetal monitor, a number of additional clinical applications can be addressed, including monitoring in labor, monitoring of high-risk patients, monitoring in transit, antepartum nonstress testing, and monitoring in the bath.

Monitoring in Labor. The technology used in the HP Series 50 T ensures that the product can be used in the very earliest stages of labor, before the membranes have ruptured, right up to and during the second stage of labor when the baby is being delivered. This means that the patient is free to move around from the onset of labor, while a reliable, continuous fetal trace is available for overseeing fetal well-being. Allowing the patient to walk around can be beneficial for the patient, especially when the delivery is long, and can even help reduce the pain of her contractions.

Monitoring of High-Risk Patients. When a high-risk patient has been admitted to the hospital for observation before the birth of her baby, it is often desirable to provide continuous monitoring of the baby to ensure its well-being. However, this is not normally practical because it would mean connecting the patient to a fetal monitor and confining her to bed for long periods of time. Using the HP Series 50 T fetal telemetry system, the patient is free to walk around, and the nursing staff has a constant overview of fetal well-being.

Monitoring in Transit. Besides being compatible with all HP fetal monitors produced since 1982, the HP Series 50 T fetal telemetry system can use the standard transducers of the HP Series 50 family of fetal monitors. This is useful, for example, if an emergency occurs and the patient needs to be transported to the operating room for a Caesarean section. In certain countries it is a legal requirement to provide continuous monitoring of the fetus from the time the patient leaves her room to the delivery of the baby. By disconnecting the transducers from the fetal monitor and connecting them to the transmitter of the HP Series 50 T, continuous, uninterrupted monitoring of the fetus is ensured.

Antepartum Nonstress Testing. Nonstress testing is performed during the patient's regular visit to the clinic or hospital during her pregnancy. By allowing the patient to ambulate and record the fetal heart rate via ultrasound, nonstress testing can be performed without having to confine the patient to bed, thereby allowing her freedom of movement and the ability to socialize with the other patients.

Monitoring in the Bath. It is becoming more common for a patient to be given a bath during labor to help reduce the pain of her contractions. Before the introduction of the HP Series 50 T fetal telemetry system, there was no safe way of providing a continuous overview of fetal well-being while the patient relaxed in the bath. This meant vital information on the fetus could be missed. By using the HP Series 50 T in conjunction with the standard watertight "blue" ultrasound and TOCO† transducers from the HP Series 50 family of fetal monitors, the nursing staff can be assured of a continuous recording of fetal information even when the patient decides to take a bath.

Fetal Monitoring Measuring Methods and Principles

The parameters measured in fetal monitoring applications are fetal heart rate, fetal movements, and maternal labor activity.

The fetal heart rate is continuously monitored on a beat-to-beat basis and recorded together with the maternal uterine activity and optionally the fetal movements on a fetal trace recorder.

There are two established methods to measure the fetal heart rate. One is to process the fetal ECG by using a fetal scalp electrode and measuring the time distance between two QRS complexes. This method is invasive and can only be used if the membranes have ruptured and the fetal scalp is accessible to attach the scalp ECG electrode. This is true only for the last few hours before delivery.

The second method of measuring fetal heart rate is to calculate the heart rate from a Doppler-shifted ultrasound signal by measuring the time distance between two signal complexes resulting from fetal heart motion. A 1-MHz pulsed ultrasound signal is emitted towards the fetal heart and the ultrasound waves are Doppler shifted by the moving parts of the heart and reflected. The reflected and Doppler-shifted signal is received again and demodulated by a 1-MHz clock signal. After filtering and amplification (by approximately 80 to 106 dB), only the low-frequency Doppler-shifted signals in the range of 100 to 500 Hz remain. These signals are fed to a loudspeaker to give the user audible feedback about the correct transducer positioning. Compared to the relatively simple algorithms needed for the heart-rate calculation from the ECG signal (the ECG signal is a well-defined, easy-to-recognize signal) the algorithm for the ultrasound signal is much more complex. The ultrasound Doppler signals contain a lot of different pulses as a result of reflections from different moving parts of the heart during one heart period. These pulses change their shapes and amplitudes depending on the angle between the ultrasound beam and the heart. Therefore, a simple peak-searching algorithm cannot accurately calculate the fetal heart rate from the ultrasound Doppler signal on a beat-to-beat basis, so a more complex algorithm using the autocorrelation function of the ultrasound signal is used. The autocorrelation function determines the similarity between all the pulses of two consecutive heartbeats. The distance between two points of highest similarity is then used to calculate the actual fetal heart rate. This method reaches a heart rate trace quality comparable to that of a trace derived from an ECG signal (the ECG signal is recognized as the "gold" standard for fetal heart rate monitoring). The advantage of the ultrasound Doppler method is that it is a noninvasive method and can be used from the twentieth week of gestation up to delivery and no direct access to the fetus is necessary.

Fetal movements can also be detected from the ultrasound Doppler signal. The fetal movement signals differ from the Doppler heart rate signal in that they have a much higher amplitude and a lower frequency. The higher amplitude is because of the bigger size of the moving areas (e.g., the fetal arms and legs) and the lower frequency is because of the lower velocity of the fetal movements compared with those of the fetal heart.

For measuring maternal labor activity (uterine activity), there are two established methods. The IUP (intrauterine pressure) method measures, as the name implies, the absolute pressure in the uterus by inserting a pressure transducer into the uterine cavity. This can be a precalibrated pressure sensor mounted in a catheter tip or a saline-solution-filled catheter with a pressure sensor connected outside. This method is invasive to the mother and can only be used if the membranes are ruptured. The second method is an external noninvasive method (external TOCO) which measures the relative hardness of the abdominal wall and the underlying uterine muscle. This method provides relative values and not absolute pressures like the IUP catheter. The pressure sensor in both transducers is based on a resistive bridge with four pressure-sensitive elements. The bridge gives a high pressure sensitivity but needs differential excitation and a differential signal amplifier.

Wireless Data Transmission

There are many possibilities for wireless data transmission from one location to another. Each method has its individual advantages and disadvantages when analyzed for a specific application. We looked at infrared and radio frequency transmission and evaluated their advantages and disadvantages for the fetal telemetry application.

† TOCO is an abbreviation for tocograph or tocodynamometer, an instrument for measuring and recording the expulsive force of uterine contractions in labor. It is usually written in uppercase letters like the abbreviations for the other fetal measuring methods: US = ultrasound, ECG = electrocardiogram, IUP = intrauterine pressure.

Infrared light is widely used as a data transmission method because of its simplicity and the fact that no regulatory approvals are necessary. However, for the fetal telemetry application, its use is not possible because the transmitting range is very limited and secure data transmission is only possible on a line-of-sight basis. This means that the transmitter cannot be covered by clothes or a bed cover and the transmission range is limited to one room. These conditions cannot be guaranteed during labor and delivery because the patient can walk around and change her position freely. Another disadvantage from the technical standpoint is the relatively high power consumption of an infrared system when used in a continuous transmission mode, which is necessary for continuous monitoring.

Radio frequency transmission, another very widespread transmission method, overcomes most of the problems of infrared transmission when it is designed carefully and an appropriate frequency range is chosen. Frequencies below 100 MHz will result in large antenna dimensions (the wavelength is 3 meters at 100 MHz) if high efficiency is needed (this is a strong requirement because of the battery-operated transmitter). On the other hand, frequencies above 1 GHz result in a wavelength (<30 cm) at which antennas become more and more directional, signal generation requires more space and power, and it takes special processes to build printed circuit boards that can handle such high frequencies.

A major disadvantage of radio frequency systems is that individual approval for each country is required and many different requirements and boundaries are given by all the national laws. These requirements must be fulfilled to obtain country approvals and should be covered by one design to avoid many special product options. Thus, the resulting design must meet the most stringent specification of all the different national standards for each requirement. A positive aspect for Europe is the upcoming harmonization within the European Community (EC) where one standard will be used for all European community members. At the moment, Germany, France, and Italy have converted this standard into national law (others will follow—a limit of two years is given for all countries to convert this standard into national law). This means that for these countries only one standard is valid.

The decision was made to use radio frequency data transmission.

The following items and specifications have been set up for a telemetry design to meet all the requirements for worldwide use:

- The frequency must be configurable in the range of 405 MHz to 512 MHz.
- The radio frequency (RF) power must be adjustable in the range of 1 mW to <10 mW.
- The spurious emissions must be <-36 dBm for frequencies <1 GHz and <-30 dBm for frequencies >1 GHz worldwide, and must be <-54 dBm in Europe in the frequency ranges 42 to 68 MHz, 87 to 118 MHz, 162 to 230 MHz, and 470 to 862 MHz.
- The RF bandwidth must be <25 kHz worldwide and should be <12.5 kHz for Japan (25 kHz would be acceptable but is not preferred)
- The transmitter must be capable of sending a special identification code after power-up for Japan.
- The RF stability over temperature (-10 to $+55$°C) and humidity (5 to 95% R.H.) must be better than ± 3.5 kHz for the U.S.A. and better than ± 2.5 kHz for Europe and Asia.

However, to design and build a radio frequency transmitter that meets all the above specifications requires extensive engineering manpower, testing, and design iterations. Therefore, we decided to reuse an existing RF transmitter and receiver for our fetal telemetry application.

After examining all possibilities, we found a good candidate in the RF parts of the HP M1400 adult ECG telemetry system. The RF parts of this telemetry system had all the approvals needed, and its highly modular design (RF parts were strictly separated from the application-specific elements) allowed us to pick up only those parts needed for our fetal application. The only modification needed was a small adaptation of the receiver's digital control software (which provides automatic frequency control—AFC—and the bitstream recovery of the digital protocol used to transfer the ECG waves). This software had to be modified to execute only the AFC function when used for the fetal application and not the bitstream recovery. It was even possible to modify the software so that it automatically switches to the correct application so that the same software can be used for the adult telemetry system and the Series 50 T. By reusing the adult RF parts, we dramatically reduced the engineering effort and only had to concentrate on the bandwidth specification (which is mainly determined by the modulation) and the special Japanese ID code.

The reused parts are the voltage-controlled crystal oscillator (VCXO) in the transmitter and as the local oscillator on the receiver side, the line amplifier as a receiver RF preamplifier, the receiver module, and all antenna parts (antennas, combiners, splitters, power tees).

Data Transmission and RF Modulation

To get low adjacent channel emission, the -6-dB RF bandwidth should not be wider than ± 8 kHz for a 25-kHz channel spacing. This gives a margin of ± 4 kHz for frequency drifts caused by temperature, humidity, and aging.

The adult ECG telemetry system uses digital Gaussian minimum shift keying frequency modulation (GMSK-FM) with a 9600-bit/s data rate. The resulting -6-dB RF bandwidth is approximately ± 8 kHz.

Because the processing power needed to calculate the heart rate from the ultrasound Doppler signal is not available in the telemetry transmitter, the ultrasound Doppler signal is transmitted to a receiver, which feeds it to a connected fetal monitor, which does all the signal processing. The telemetry system only acts as a wireless analog front end for the fetal monitor. The HP Series 50 fetal monitors sample the signals at a 1.6-kHz sample rate with 12-bit resolution. To transmit the ultrasound signal as a digital bitstream, the required data rate is 12 bits $\times$ 1600 samples/s = 19200 bits/s for the ultrasound signal alone. Together with the uterine activity signal and the necessary framing and checksum overhead a minimum data rate of 22 kbits/s is required. This data rate does not include any redundancy needed for error correction.

To fit into the 25-kHz channel bandwidth, this data rate must be compressed to 9600 bits/s. This requires highly sophisticated data compression circuitry. The data stream resulting from a sampled ultrasound Doppler signal does not contain as much redundancy as the ECG, which does not change rapidly except for the short duration of the ECG QRS pulse. To fit into a 12.5-kHz channel spacing an additional data reduction down to 4800 bits per second is needed.

We decided to transmit the heart rate signal (ultrasound Doppler or ECG) with standard direct FM modulation. The uterine activity signal together with some status signals—battery status, nurse call function, serial number, and transducer modes (ultrasound or ECG and external TOCO or IUP)—are transmitted as a digital bitstream. This information is transmitted four times per second. Every data block is secured with an 8-bit checksum (CRC). A data block always starts with the serial number of the transmitter. This serial number,which is the same for the transmitter and the corresponding receiver, is used by the receiver to synchronize itself with the datastream and to verify that the data is coming from its own transmitter. This ensures that signals from two different patients using the same RF frequency are not mixed. The overall data rate for the digital transmitted signals is 200 bits/s. This data stream is transmitted as a frequency shift keying (FSK) signal with a 1600-Hz signal for a logic 0 and a 2400-Hz signal for a logic 1. This signal, added to the heart rate signal, frequency modulates the RF carrier. The amplitude of the composite signal determines the required RF bandwidth and can easily be adapted to meet the 12.5-kHz channel spacing requirements.

RF Bandwidth

The resulting RF bandwidth can be estimated as follows. The modulation signal is composed of two components: (1) the ultrasound Doppler signal or the ECG signal and (2) the FSK subcarrier signal. The modulation spectrum is illustrated in Fig. 2.

The RF FM modulator has a sensitivity of 1.6 kHz/V, which is the specification of the reused RF oscillator from the adult ECG telemetry system. The ultrasound signal has a bandwidth BW_{lf} = 500 Hz and an amplitude of 1.875 V_{p-p} which produces an RF carrier shift of 1.6×1.875 = 3.0 kHz. The corresponding modulation index is β = frequency shift $\div$ modulating frequency = 3.0 kHz/500 Hz = 6. The ECG signal has a bandwidth BW_{lf} = 100 Hz and a carrier shift of 3 kHz, so the modulation index is β = 3.0 kHz/100 Hz = 30.

The FSK signal has as its highest frequency a 2.4-kHz sinusoidal carrier. Its amplitude produces an RF carrier shift of 1.5 kHz. The modulation index is β = 1.5 kHz/2.4 kHz = 0.625.

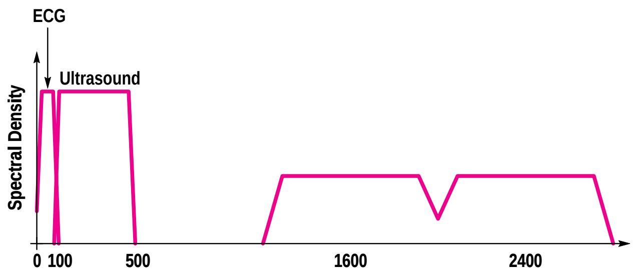


Fig. 2. Modulation signal spectrum of the HP Series 50 T fetal telemetry system transmitter.

For a modulation index less than one the RF bandwidth is approximately $BW_{rf} = 2BW_{lf}$. Only the Bessel functions of orders 0 and 1 have significant values (>0.01), so they represent 99% of the RF energy, or in other words, outside this bandwidth the signal is 20 dB down from the maximum at center frequency. Thus, the −20-dB RF bandwidth of the FSK carrier is 2×2.4 kHz = 4.8 kHz. For a bandwidth where the signal is 40 dB down (99.99% of the RF energy is within this bandwidth) the Bessel function of order 2 is also of interest and the −40-dB RF bandwidth of the FSK carrier is 4×2.4 kHz = 9.6 kHz.

For a modulation index greater than one, the RF bandwidth is approximately $BW_{rf} = 2(\beta+1)BW_{rf}$ for a bandwidth where the signal is 20 dB down. For a bandwidth where the signal is 40 dB down this bandwidth doubles again: $BW_{rf} = 4(\beta+1)BW_{lf}$.

With a modulation index of 6, the ultrasound signal produces an RF bandwidth of BW_{rf} (−20-dB) = 2(6+1)500 Hz = 14×500 Hz = 7 kHz or BW_{rf} (−40-dB) = 4(6+1)500 Hz = 28×500 Hz = 14 kHz.

With a modulation index of 30, the ECG signal has an RF bandwidth of BW_{rf} (−20-dB) = 2(30+1)100 Hz = 6.2 kHz or BW_{rf} (−40-dB) = 4(30+1)100 Hz = 12.4 kHz.

Thus, the overall RF bandwidth is mainly determined by the ultrasound signal or the ECG signal and not by the FSK signal.

The amplitude ratio between the heart rate signal and the FSK signal was chosen so that as the field strength at the receiver input goes down, the signal-to-noise ratio of the FSK signal decreases before the heart rate signal is affected. The receiver detects bit errors in the digital data stream and suppresses the heart rate output signals to the fetal monitor when errors occur. Thus, the fetal monitor always shows either the correct heart rate values or no value, but never displays wrong values, which may lead to a misdiagnosis.

Telemetry Transmitter

Fig. 3 shows the components of the HP Series 50 T fetal telemetry system.

A high priority for the telemetry system design was to support the same transducers as used by the HP Series 50 fetal monitors. Customers can use the transducers they normally use with their fetal monitors and can switch between standard monitoring and telemetry monitoring simply by replugging the transducer connectors from one device to the other. Repositioning and reapplying the transducers on the patient are not necessary and the switch can be performed in a few seconds. This compatibility was no problem with the ultrasound and uterine activity transducers, but the fetal monitor ECG transducer required more detailed investigation to design circuitry to handle this transducer in the telemetry transmitter.

The HP M1357A fetal ECG transducer is an active transducer in which the complete ECG preamplifier and its floating power supply are incorporated in the transducer legplate. Since the telemetry transmitter is battery powered, this floating, highly isolated preamplifier is overdesigned for

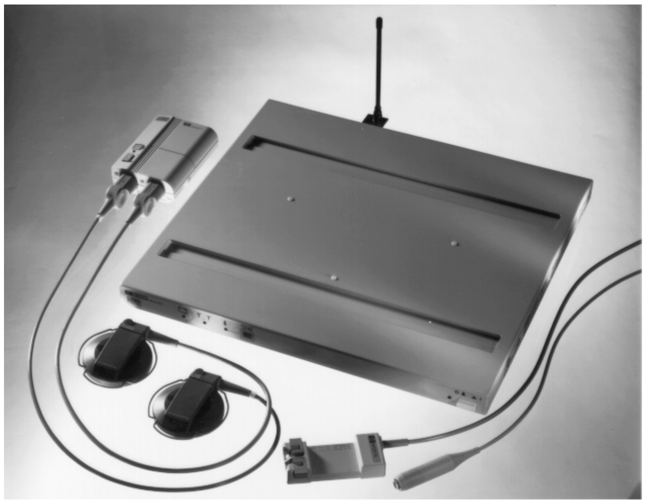


Fig. 3. Transmitter, transducers, and receiver of the HP Series 50 T fetal telemetry system.

telemetry use. However, it is mandatory for use on a mains-powered fetal monitor, since patient safety requires all transducers that have direct contact with the patient via electrical conducting electrodes to be floating.

The M1357A transducer requires a 10V peak-to-peak power supply signal with a frequency between 100 and 250 kHz. The ECG signal is transferred on the same wires by power load modulation, which means that the transducer varies its load on the driving circuit with the amplitude of the ECG signal. A circuit had to be designed that is capable of driving the HP M1357A transducer with a 10V peak-to-peak signal at 250 kHz, sensing the load current, and operating from a 5V supply. A bridge driving circuit built with digital 74AC14 inverters running at 250 kHz was found to be capable of delivering the required drive signal with enough power to supply the transducer. The load current is sensed by a 5-ohm resistor in the ground connection of the 74AC14 drivers. Fig. 4 shows the ECG transducer driver circuit.

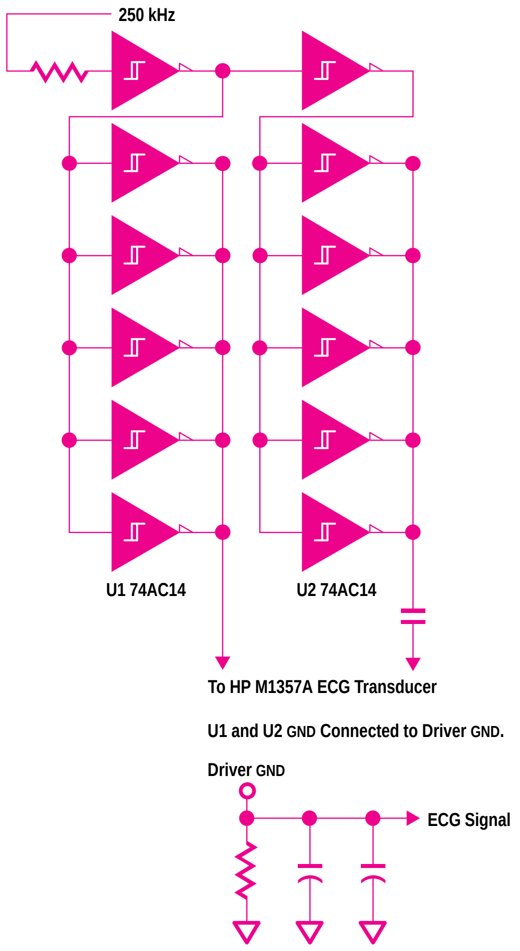
A major consideration when designing a telemetry system is the power consumption of the transmitter. It runs from batteries and therefore a goal is to make the operating time as long as possible with one set of batteries. The low-power design of the HP Series 50 T transmitter extends the operating time to more than 40 hours of continuous operating with ultrasound and external TOCO transducers connected. The power source is three AA alkaline cells. This is two to three times longer than competitive fetal telemetry systems. When used with NiCd batteries, the operating time is comparable with competitive systems, but the weight of the HP Series 50 T transmitter is only half that of the lightest competitive transmitter.

In addition to weight and operating time, another major aspect of telemetry system design is transmitter size. To get the best use of the available volume, the HP Series 50 T transmitter uses double-sided surface mount technology on the printed circuit boards. This reduces the board space by 40 to 50% compared to single-sided technology and makes the HP Series 50 T transmitter the smallest and lightest of all competitive fetal telemetry transmitters.

Fig. 5 is a block diagram showing all major functions of the transmitter. A microcontroller is the heart of the transmitter. This seemed to be the best solution, considering all of the required features such as Japanese ID code transmission after power-up, nurse call function, serial number handling, analog hardware control depending upon the type of transducer (ultrasound or fetal ECG, TOCO or IUP), battery status, and CRC calculation for every data frame. The microcontroller gives more flexibility than an ASIC and the development time was shorter. With an appropriate controller it is possible to execute the fetal movement detection algorithm in the transmitter, thereby saving transmission capacity of the RF channel by transmitting only the movement detection bit instead of the fetal movement Doppler signal.

Microcontroller Features

We chose the Mitsubishi M37702-M2 16-bit controller. This controller has true 16-bit processing power and many integrated peripheral functions, and is low in cost. It has 512 bytes of internal RAM and 16K bytes of ROM. There are eight independent 16-bit timers. Five of these have their inputs and outputs accessible on pins, can individually select the input clock from a predivider, and can run in several modes including timer, counter, pulse width modulator, one-shot, free-running, or triggered from input pins or software. The controller also has an independent watchdog timer,



250 kHz
U1 74AC14
U2 74AC14
To HP M1357A ECG Transducer
U1 and U2 GND Connected to Driver GND.
Driver GND
ECG Signal

Fig. 4. HP M1357A ECG transducer driver.

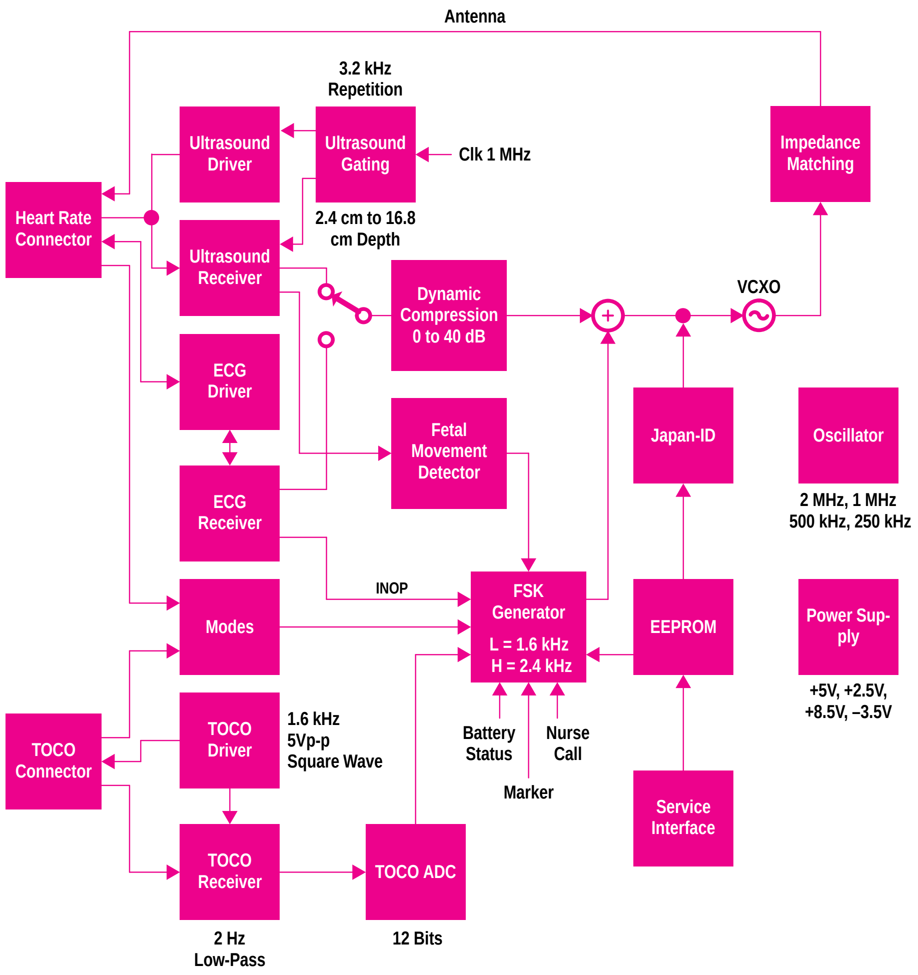


Fig. 5. HP Series 50 T fetal telemetry system transmitter block diagram.

eight channels of 8-bit analog-to-digital converters, and two independent synchronous or asynchronous serial communication channels. The package is a plastic quad flatpack with 80 pins. The rich set of integrated peripherals on the controller chip allowed us to save a lot of hardware that would otherwise be needed outside the controller.

The controller is available with 8-MHz, 16-MHz, or 25-MHz input clock speed. The processing power needed in the HP Series 50 T transmitter allows a reduction of the clock frequency to 2 MHz. When running with the 2-MHz input clock and all peripheral functions active, the M37702 controller consumes only 1.5 mA with a 5V power supply.

Three timers (one in timer mode, two in one-shot mode) are used to produce the gating signals for the pulsed ultrasound Doppler channel. Fig. 6 shows the resulting gating signals. With a sound velocity of 1500 m/s in human tissue, the resulting ultrasound sensitivity over depth can be calculated. The minimum depth is determined by the delay time between the end of the ultrasound transmit pulse and the start of the receive gate pulse, which is t_3 in Fig. 6. With $t_3 = 32$ µs, $d_{min} = (1500 \times 10^3 \times 32 \times 10^{-6})/2 = 24$ mm. The maximum depth is determined by the time between the start of the transmit gate and the end of the receive gate, which is $t_2 + t_3 + t_4 = 224$ µs. The depth is then $d_{max} = (1500 \times 10^3 \times 224 \times 10^{-6})/2 = 168$ mm. The factor of 2 in these calculations results from the fact that the ultrasound wave propagates first towards the reflecting object located at depth d and then back again to the transducer.

Two timers are used to produce clock signals needed in the ECG amplifier and the TOCO transducer excitation circuitry. One timer is used to produce the 1600/2400-Hz FSK signal. One timer in count mode, together with an external first-order sigma-delta modulator (one comparator and one flip-flop), forms a 12-bit analog-to-digital converter for the uterine activity signal.

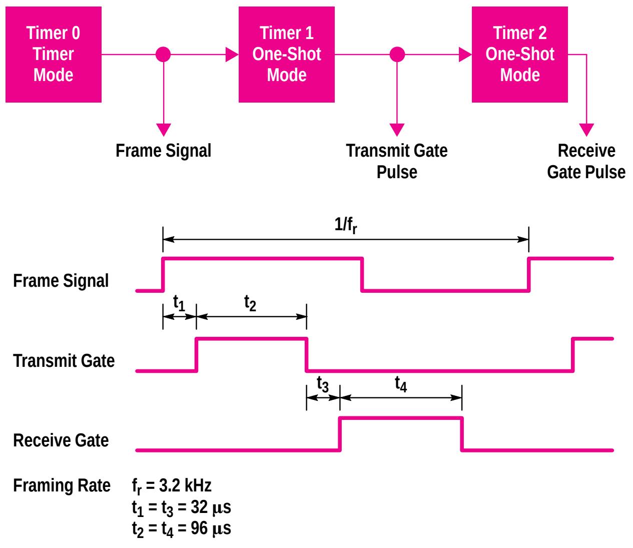


Fig. 6. Ultrasound Doppler gating.

One serial communication channel in synchronous mode is used to control a serial EEPROM to store the serial number, some calibration constants, the country option codes, and the power-up ID code for Japan. The second serial communication port is used as a production and service port to read out internal signals, to program the EEPROM, and to send messages during power-up if self-test failures are detected.

The A-to-D converters are used to monitor the battery voltage, to measure the ultrasound Doppler or ECG signal amplitude to control the signal gain, to measure the fetal movement signal amplitude, and to measure some test voltages during the power-up self-tests.

Uterine Activity Measurement Circuitry

The uterine activity transducers are built with four pressure-sensitive resistors in a bridge configuration. This bridge configuration requires a differential excitation voltage and a differential sensing amplifier. The bridge resistors are in the range of 300 to 1000 ohms. The requirement of compatibility with standard fetal monitor transducers did not allow the use of a new transducer with a higher impedance to reduce the drive power. The IUP transducers are active and need a drive voltage of at least 5Vdc or 5Vac at >1 kHz. The resulting power consumption for the 300-ohm type is then $P = V^2/R = 25/300 = 83.3$ mW.

A circuit can be designed that reduces this power level to 45 mW. The disadvantage is a sensitivity reduction by 6 dB, that is, the sensitivity is halved. This can be compensated by doubling the gain of the sensing amplifier. A 5Vac excitation at 1.6 kHz (half the repetition frequency of the pulsed ultrasound Doppler to avoid interference between the TOCO drive circuitry and the ultrasound demodulator) was chosen instead of a simpler dc drive circuit because low-power operational amplifiers running on 5V or less with high dc precision have not been available for reasonable prices. The labor activity signal, which is a signal with a bandwidth from dc to <2 Hz, is obtained by a synchronous demodulator running at 1.6 kHz and a low-pass filter. By using ac excitation, the sense amplifier can be built with simple and inexpensive TL062A amplifiers.

Fig. 7 shows the implementation of this circuitry. The TOCO excitation applies power (+5V and ground) for the first half of a 1600-Hz square wave (50% duty cycle). During the second half, the excitation drive is switched off. This halves the power consumption of the TOCO transducer resistive bridge. A differential amplifier senses the bridge signal, amplifies it, and converts the differential signal into a single-ended signal. The input capacitors settle to the bridge output voltage during the active drive phase of the excitation driver and discharge to a middle value during the nondriving phase through the bridge resistors, In this way, the sensing amplifier input picks up a 1600-Hz signal with half the amplitude compared to a full-bridge excitation driver. The signals are illustrated in Fig. 8.

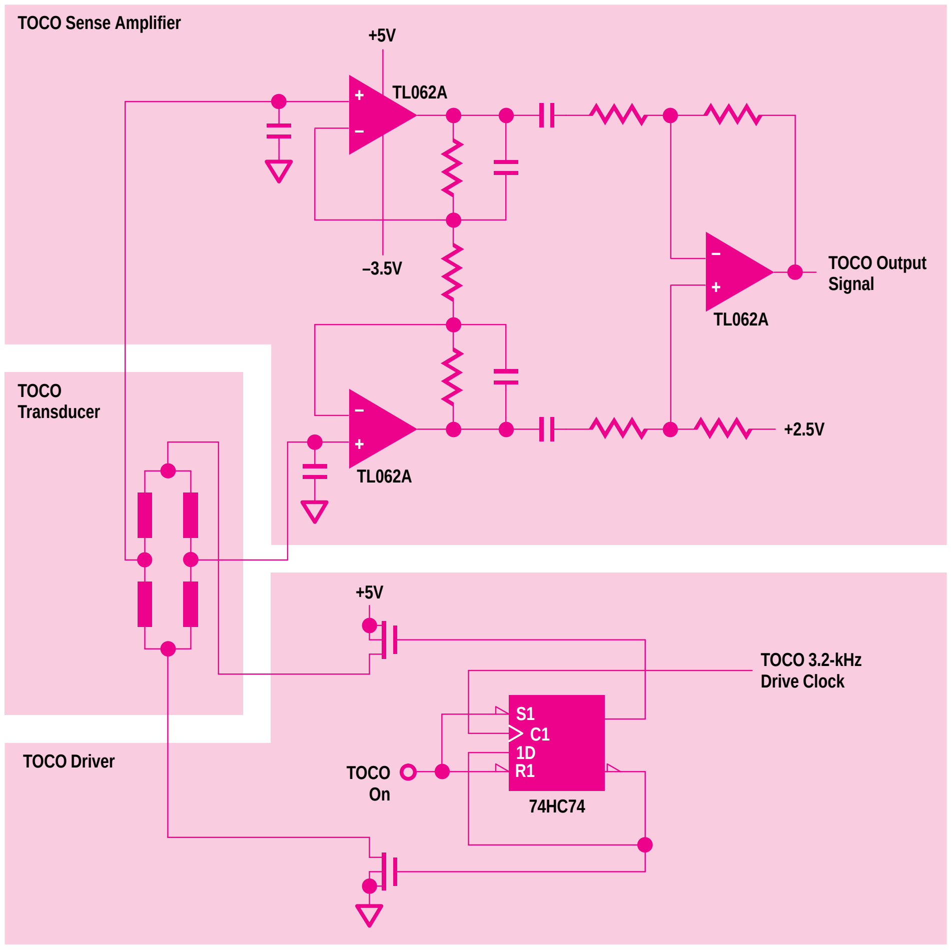


Fig. 7. TOCO driver and sense amplifier.

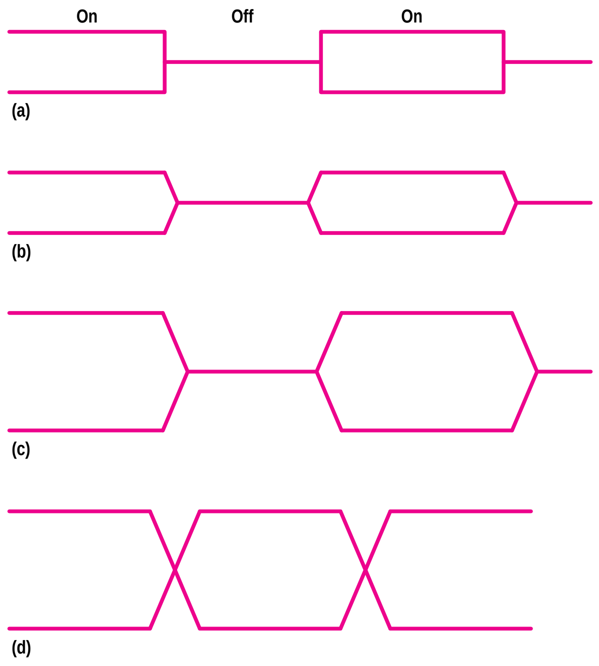


Fig. 8. TOCO signal waveforms. (a) Excitation signal. (b) Sense amplifier differential input (low bridge differential signal). (c) Sense amplifier differential input (high bridge differential signal). (d) Sense amplifier differential input with true bridge excitation.

Power Supply

An essential part of a battery powered handheld device is the power supply. The fetal telemetry transmitter is designed to run with three AA-size alkaline batteries or rechargeable NiCd accumulators. The new and more environmental friendly NiMH accumulators are also supported.

The input voltage is from 2.5 to 4.8 volts. The power supply must provide the following output voltages:

- +5V. Main supply voltage for all digital and most analog circuits
- +2.5V. Virtual ground for 5V single-supply analog circuits
- +8.5V. Supply for some operational amplifiers and the ultrasound receiver preamplifier
- −3.5V. Negative supply for a few operational amplifiers to increase the signal dynamic range.

The power supply is a switched-mode type delivering a stable (±1%) +5V output from the batteries. All other voltages, which need only a few milliamperes, are built with charge pumps or simple buffered voltage dividers. The switched-mode power supply runs at 250 kHz in a pulse width modulation mode. The 250-kHz switching frequency has the advantage that only small inductors are needed. Part of the power supply is also the main crystal-controlled clock oscillator running at 4 MHz. All other clock signals are derived from this master clock. The oscillator and the power supply are designed to start with input voltages as low as 2.0V. The efficiency of the power supply varies between 70% for low (2.5V) input voltages and 82% with a 4.5V input. Fig. 9 is a diagram of the transmitter power supply.

Japanese ID Code

Japanese radio frequency laws require that a special identification code be transmitted every time the transmitter is switched on. The code bitstream is modulated on a subcarrier with a speed of 1200 or 2400 bits/s or is direct FSK or GMSK modulation of the radio frequency carrier signal at 1200, 2400, or 4800 bits/s for FSK and 2000, 4000, or 8000 bits/s for GMSK modulation. The bit rate must be accurate within a tolerance of ±200 ppm. The code is composed of (1) >100 bits of alternating ones and zeros, (2) a 31-bit maximum-length pseudorandom noise code sequence, (3) 51 ID code bits (provided by the regulatory agency, this code is unique for every transmitter and contains information about the device manufacturer and the product, and a unique serial number that has nothing to do with the normal product serial number), (4) a 12-bit checksum calculated from the 51 code bits by a special polynominal division.

In the HP Series 50 T transmitter, this code is stored in the EEPROM during the production final test. During a power-up sequence, this code is read by the transmitter microcontroller and transmitted as a 1200-bit/s FSK signal before starting normal transmission. The code in the EEPROM is also secured by a checksum. If this checksum is corrupted, the transmitter will not start normal transmission as required by the regulatory agency. This feature is only active for Japanese options (also stored in the EEPROM) and is ignored for all other countries.

Modulation Circuits

The modulation circuits have a twofold responsibility. One is controlling the RF bandwidth with its amplitude and frequency characteristics and thus maintaining conformity with RF regulatory requirements. The other is making the best use of the available RF bandwidth to get the best possible signal-to-noise ratio for the transmitted signals.

Fig. 10 shows the modulation circuits. The circuitry consists of three subcircuits: a programmable-gain amplifier, a limiter, and a low-pass FSK shaping filter.

The programmable-gain amplifier adjusts the heart rate signal amplitude to a value that corresponds to 60% of the maximum allowable RF bandwidth. The margin of 40% is to accommodate the often rapidly changing signal amplitude, especially for the ultrasound Doppler signal. The gain of the amplifier is controlled by a regulator algorithm implemented in the M37702 controller. The current signal amplitude is measured with one of the integrated A-to-D converter channels, and from this value an appropriate gain is calculated for the programmable-gain amplifier. This amplifier consists of an operational amplifier with an 8-bit multiplying D-to-A converter in its feedback path. So as not to change the gain too much during one heart period (which could lead to a wrong heart rate calculation for the affected beat), the new gain value is adjusted linearly over two seconds. Therefore, the 40% margin is provided so that the amplifier does not overdrive the signal too often.

The limiter circuit following the programmable-gain amplifier clips the signal to well-defined limits in the case of a suddenly

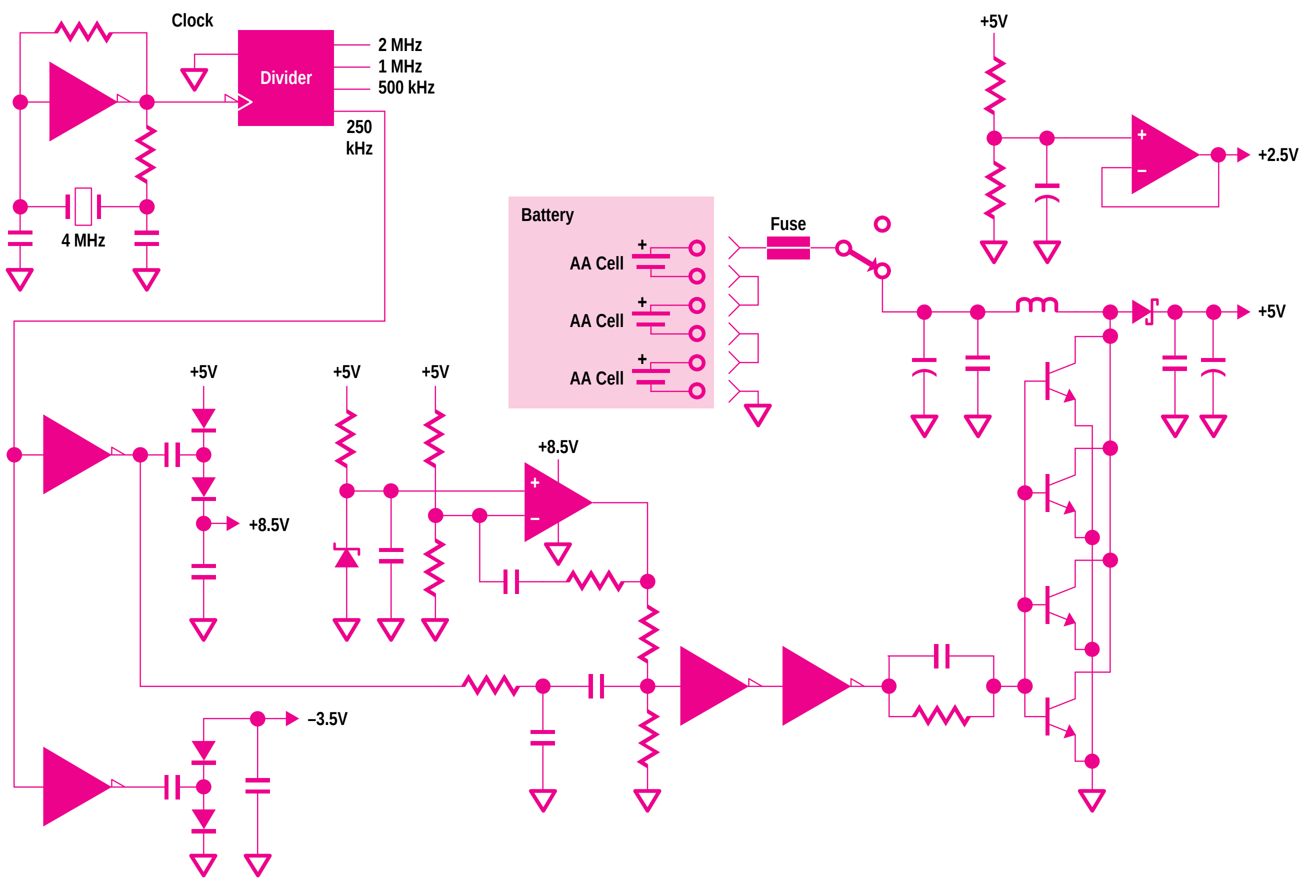


Fig. 9. Transmitter power supply.

increasing input signal to the programmable-gain amplifier. The low-pass filter, which also acts as a bandpass filter and a summing amplifier for the square wave FSK signal, removes the overtones resulting from the clipping and thereby ensures that the modulation has a well-defined RF bandwidth.

Telemetry Receiver

The recovery of the digital bitstream in the FSK signal is the main task of the receiver. The FSK signal is extracted from the composite signal (FSK and heart rate signal) by an analog bandpass filter with 1400-Hz and 2600-Hz corner frequencies. The recovered sinusoidal FSK signal is converted into a square wave by a comparator. All subsequent recovery tasks are implemented in a Mitsubishi M37702-M2 microcontroller (the same type as used in the transmitter).

Digital data recovery can be divided into two main tasks: FSK signal demodulation (recovering the single bits from the 1600/2400-Hz input signal) and synchronization with the

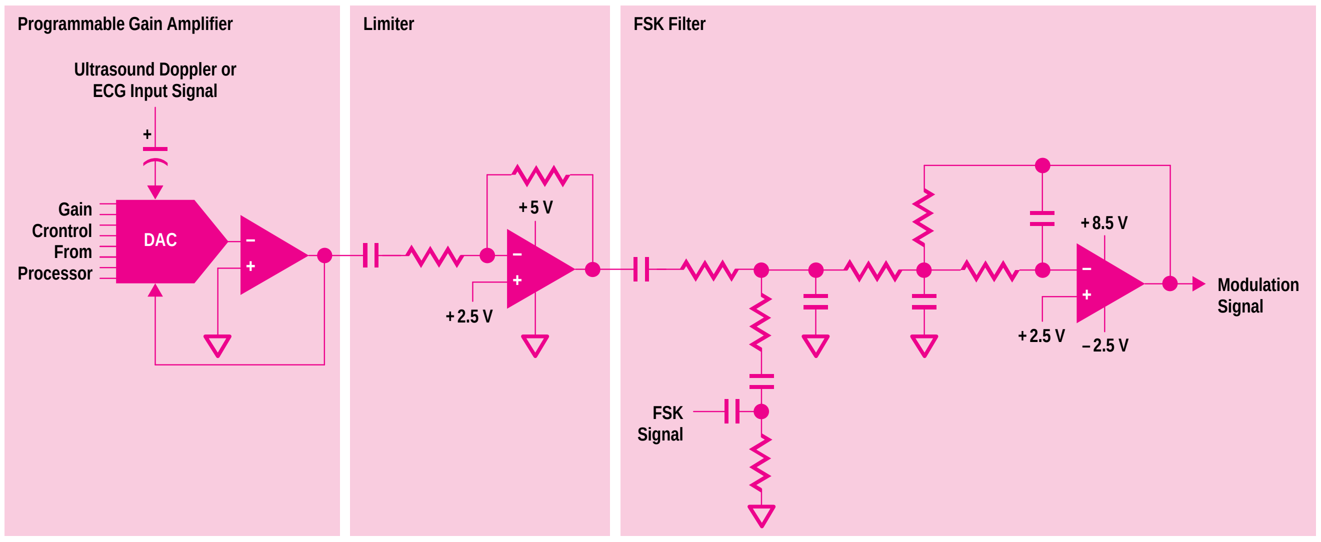

Programmable Gain Amplifier
Ultrasound Doppler or ECG Input Signal
Gain Crontrol From Processor
DAC
Limiter
+5 V
+2.5 V
FSK Filter
+8.5 V
+2.5 V
−2.5 V
Modulation Signal
FSK Signal

Fig. 10. Transmitter modulation circuits.

transmitted data frames. Fig. 11 shows the FSK signal demodulation scheme, which is completely implemented in the microcontroller.

The pulse period measuring circuitry is composed of two microcontroller timers. The first timer is configured as a retriggerable 250-µs one-shot and the second is configured for pulse period measurement. The incoming FSK square wave signal first triggers the one-shot, which suppresses very short periods between two positive edges in the incoming signal and triggers the pulse period measurement timer. The timer generates an interrupt whenever a new pulse period measurement is complete. For noise rejection, the period values are low-pass filtered and all period values that are outside the limits for a 1600-Hz or 2400-Hz input frequency are rejected. This greatly improves the signal recovery for noisy input signals.

The integrate and dump block sums all of the incoming period values. After five milliseconds, which is one bit time, the sum is reset by a dump signal delivered by the bit clock recovery digital phase-locked loop. Before reset, a comparator decides if the received bit was a logic one or a zero.

The phase-locked loop block recovers the bit clock from the incoming data stream and generates the sample and dump signals, which are phase synchronized with the incoming data clock. Only input sequences consisting of one or two equal bits (bit patterns 010, 101, 0110, and 1001) are used for the phase tracking. The one- or two-bit detector produces a reference signal whenever one of the four bit patterns is detected in the incoming bitstream. The detector measures the time between changes in the incoming period signals and checks to see if this time falls within the limits for one or two bit times (4 to 6 ms for one bit and 9 to 11 ms for two bits).

To extract the individual frames from the recovered bitstream, the structure shown Fig. 12 is used. The recovered bits are shifted into a 50-bit-deep serial in, parallel out shift register, which is the length of one frame. A complete frame can be stored in this shift register and all of its bits analyzed at once.

The header search algorithm looks for a valid header pattern in bit positions 36 to 48. The header pattern is derived from

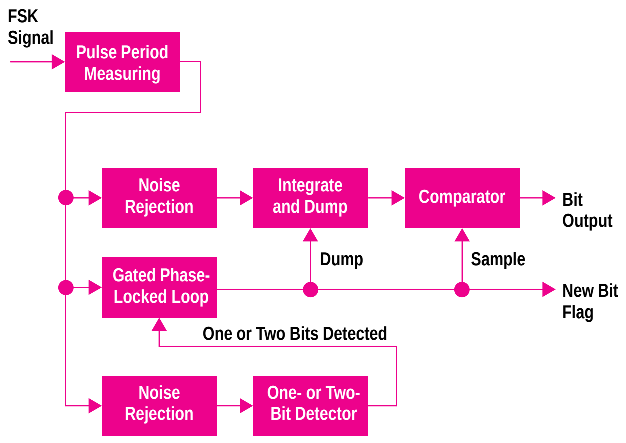


Fig. 11. FSK signal recovery scheme implemented in the HP Series 50 T fetal telemetry system receiver.

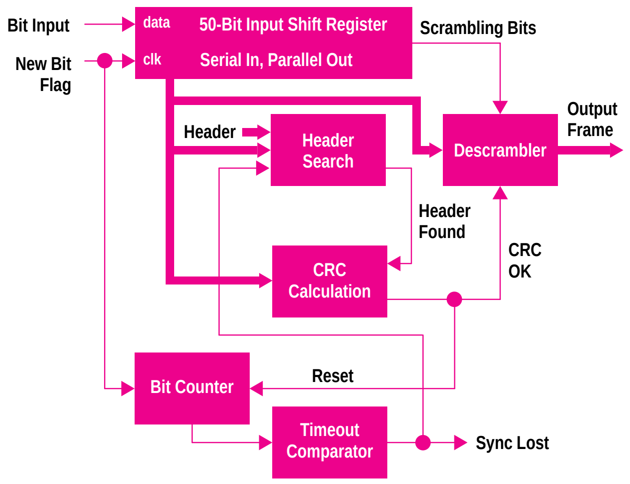


Fig. 12. Data stream recovery in the HP Series 50 T fetal telemetry system receiver.

the system serial number (transmitter and receiver have the same serial number stored in their EEPROMS). Bits 49 and 50 are used to identify whether a frame has been scrambled before transmission. The scrambling is done if the original frame did not contain enough single or double bit patterns for the clock recovery phase-locked loop. The scrambling is done by inverting every second bit in the frame.

If a valid header (original or scrambled as indicated by the scrambling identification bits) is found, the complete frame is checked for a good CRC pattern by the CRC calculation block. If the CRC is OK, the complete frame is deassembled into the original information and saved, the header search is disabled for a complete frame, and a synchronization done flag is set.

If the CRC is not OK for more than two frames, the header search is enabled again and the synchronization lost flag is set. The time since the last correct CRC is measured by a bit counter. If more than 100 bits are received without a correct checksum, the synchronization process is restarted.

Test Strategy

To ensure maximum reliability and safety, extensive self-tests are executed each time the transmitter and receiver are powered up. Not only are the easy-to-test digital parts checked, but also most of the analog hardware. This is done by generating artificial signals, feeding them into the different analog signal paths and measuring the response of each path. All these tasks are performed by the onboard M37702 controllers in the transmitter and the receiver. The results obtained are checked against limits. If any deviation is detected, a clear failure message is sent out over the integrated service port (RS-232 line) to assist in troubleshooting, and the transmitter or receiver tries to restart. This ensures that a faulty device does not go into normal operation and that no incorrect data is transmitted or displayed by the attached fetal monitor. Tests have shown that about 80% of all part shorts or opens are detectable in the analog processing parts. This is a very high number for power-up analog tests. Achieving such a high error detect rate was only possible through the use of a powerful microcontroller. The software for self-test and service support is about 40% of the complete software package.

Production testing of the transmitter and receiver is divided into two parts: single component or board testing before final assembly and final test of the completely assembled device.

The loaded boards are tested by an automatic test system. Connection to the circuitry on a board is made by a needle bed adapter which allows stimulation and measurement of every net and every component on the board. The goal of this test is to verify that all components are loaded correctly and have the right values. If a component fails, the tester reports the component, its location and the detected failure. For surface mount boards, most failures are bad solder joints and shorts between pins. To test complex parts like RAMs, A-to-D converters, microprocessors, and microcontrollers, a library model of the part describing its behavior on predefined stimuli is required. To check for good solder joints, a model that toggles every pin of a component as input or output is sufficient. Unfortunately, such a model was not available for the M37702 controller, which is an 80-pin device in a quad flat package. To test this component, we implemented special software in the controller itself, which can be activated by the test system by pulling a pin to +5V. This pin is checked by the software during the power-up cycle and the special test software is entered if the pin is pulled high. The special software mirrors the input pins to the output pins. The test system only has to apply a test pattern to the inputs and then check for this pattern on the corresponding output nets.

The HP Series 50 T fetal telemetry system uses the same final test equipment as the Series 50 fetal monitors. Most of the final specification tests are similar or identical to the Series 50 fetal monitor tests. Therefore, we implemented a production and service interface identical to those of the fetal monitors. Only a few tests had to be added to cover the telemetry-specific specifications. The test itself is highly automated and controlled by a workstation computer. This computer controls the measurement equipment, performs the measurements, prints the results and stores the results in a database. This allows continuous production process control by calculating the Cpk value (a value that describes the production process capabilities) for each test specification to check the stability of all production processes. This also makes it possible to detect test result drift resulting from part changes before a test result completely fails the specification. This process control procedure is supported by ISO 9001 and EN 46001 certification rules, and it really increases the product quality and stability, which the customer can directly see.

Support Strategy

The HP Series 50 T fetal telemetry system can be used as a standalone unit with a local antenna, or the receiver can be connected to an existing antenna system. When connected to an antenna system the coverage area is increased. The design of antenna systems and the connection of the Series 50 T to an antenna system is the responsibility of HP customer engineers. For standalone systems the system is designed to be installed by the customer (plug-and-play).

The acceptance tests needed to ensure proper functionality are built into the firmware and can easily be performed by the customer. In case of any problems the customer can call the HP medical response center. The response center engineer has the ability to give troubleshooting instructions and find the defective assembly.

All of the low-frequency assemblies can be replaced onsite or on the repair bench. The RF assemblies (400 to 500 MHz) can only be repaired on the repair bench because high-precision RF instruments are needed to do RF troubleshooting. Special service software is available to assist in troubleshooting. This software provides check data for transmission, monitors field strength, and transfers serial numbers when needed for repair.

The support features were implemented with minimal effort because the support requirements were discussed with field support personnel and their inputs were considered by R&D in the design phase.

Error Detection and Display

The HP Series 50 T is designed to show any software malfunction through the use of red LEDs in a certain sequence. Hardware failures can be troubleshot by the response center by telephone by following certain procedures and noting the results.

The processor boards of both the transmitter and the receiver run self-test routines after power-on to test hardware functionality and software integrity. After power-on, the receiver switches on all LEDs for one second to test them, and then returns to normal operation. If the power-on test fails, all LEDs stay lit, indicating an error condition. After power-on, the transmitter switches on a red LED hidden behind the positive connection on the middle battery inside the battery compartment. This LED stays on for three seconds, and if everything works it is switched off. If there are any errors this LED stays lit. All this visible information is very helpful to the response center engineer checking for system malfunctions via telephone with the customer. A defective section can be located in a very short time with high accuracy, helping to ensure low cost of ownership for the user.

Troubleshooting Tools

Troubleshooting tools are built into the system to provide an internal error log, reporting on settings and failures. This log can be accessed by software running on a standard PC via an RS-232 connection to the receiver or transmitter. The software log is detailed and includes an interpretation of each error message, so no manual is required.

Should a fetal telemetry system need repair, the software to test the internal functions and do simple troubleshooting is built into the unit. On the repair bench, during onsite repair, or during biomedical testing, it is only necessary to connect the system to a standard PC and start the service software to have the built-in troubleshooting help available. The connection between the PC and the transmitter or receiver is a 3-wire RS-232 interface. All transmitter and receiver responses can be tested. For hardware replacements, such as the transmitter or receiver CPU board, the serial number of the system needs to be written to the new board using the service software. To avoid typing errors, we decided to read the serial number from the nondefective unit (transmitter or receiver) and transfer it to the new unit (receiver or transmitter). In the case of intermittent failures the PC running the service software can be connected to the system and the PC can collect the error log overnight. The service software is designed to be totally self-describing with all reported messages interpreted, thereby avoiding error codes, which require error tables to find the problem description. The main screen of the service software shows the following information:

```
* * * * * M1310A Service Software Rev.A.01.0   * * * * *
*                  M A I N  M E N U                    *
*           ˜ Program S/N to Transmitter               *
*           ˜ Program S/N to Receiver                  *
*           ˜ Power On Selftest                        *
*           ˜ Show last errors/warnings                *
*           ˜ Check Transmitter                        *
*           ˜ Check Receiver                           *
*           ˜ Read SerNum and Revisions                *
*           ˜ Reset Serial Number                      *
*           ˜ Read country information                 *
*           ˜ EXIT                                     *
*                                                      *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
*        Select with <up>, <down>, <enter>             *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The following represents the first screen to program the serial number to the transmitter:

```
* * * * * M1310A Service Software Rev.A.01.00 * * * * *
*              Program S/N to Transmitter              *
*                                                      *
*           ˜ follow the steps <ENTER>                 *
*     = >  (1)    plug cable to RECEIVER               *
*          (2)    READ S/N from RECEIVER               *
*                 RCVR-S/N is : ..........             *
*          (3)    plug cable to TRANSMITTER            *
*                 checking XMTR-S/N                    *
*          (4)    WRITE S/N to TRANSMITTER             *
*                 XMTR-S/N is : ..........             *
*                                                      *
*           ˜ Return to MAIN                           *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
*         Select with <up>, <down>, <enter>            *
* * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

The user is guided step-by-step through the program; no manual is needed.

Installation Acceptance

After installing the fetal telemetry system the performance of the system should be tested. The installation acceptance test is built-in. Overall transmission between the transmitter, receiver, and fetal monitor is checked by creating a synthetic signal. This is a simple operation that can be done by the customer.

The synthetic signal for the acceptance test is generated in the transmitter and shows a test pattern on the fetal monitor. One heart rate transducer and one TOCO transducer can be connected to the transmitter, and the acceptance test gives the appropriate output for the transducers connected. The acceptance test is started by pushing and holding down the nurse call button while switching on the transmitter power. The test runs as long as the nurse call button is pushed. On the fetal monitor a heart rate is measured and a TOCO triangular waveform shows the proper functioning of the overall system. This acceptance test verifies the overall transmission from the transmitter to the receiver via radio frequencies and the transmission from the receiver to the fetal monitor via cable connection. If all the signals are transmitted as expected the fetal telemetry quality is acceptable.

The acceptance test is designed to avoid any need for external test tools or measurement equipment. Because it is easy to perform and no external equipment is needed, this test helps save installation costs and reduces cost of ownership.

Acknowledgments

Although it's not possible to mention everyone who contributed to the success of this project, our gratitude and thanks go to Andrew Churnside, product manager, Traugott Klein, transmitter mechanics and tooling, Siegfried Szoska, receiver mechanics, Erwin Müller, software engineering, Stefan Olejniczak, hardware and software engineering, Dietrich Rogler, design, Herbert Van Dyk, regulations, and Peter Volk, manufacturing. Thanks also to the HP M1400 adult telemetry system development team, especially Mark Kotfila, product manager and Larry Telford, software implementation. Finally, thanks to the other members of the crossfunctional team who contributed to the success of this project.

Zero Bias Detector Diodes for the RF/ID Market

Hewlett-Packard's newest silicon detector diodes were developed to meet the requirements for receiver service in radio frequency identification tags. These requirements include portability, small size, long life, and low cost.

by Rolando R. Buted

Tracking of products and services is critical in today's highly competitive and rapidly growing world of manufacturing and service industries. To succeed in these industries, accurate and timely information is required.

Two widely used tracking methods are bar code readers and magnetic stripe. Although commonplace, they are both limited in their range and their operating environment. For example, bar codes require a direct line of sight within a few inches and a relatively clean and benign environment to operate reliably.

In contrast, a radio frequency identification (RF/ID) system uses radio signals to communicate. Line of sight is not needed and the system can operate in hostile environments characterized by water, oil, paint, and dirt. It can even be used for communication through cement, glass, wood, or other nonmetallic materials. These wireless systems are being successfully used to identify and track cattle, household pets, cars passing through toll booths, supermarket carts, railroad cars, and personnel entering and leaving secure facilities.

An RF/ID system is composed of two components: a reader (interrogator), which contains both transmitter/receiver and decoder/control modules, and a tag (transponder), which typically contains an antenna and a receiver circuit. Since a system normally has only a few interrogators but many tags, the most severe design constraints are on the tag. These constraints include portability, small size, long life, and low cost. Hewlett-Packard's newest silicon detector diodes (HSMS-285x) were developed to address these constraints.

RF/ID Technology

RF/ID tags can be active or passive. Active tags have an onboard power source (a battery) so that less power is needed from the reader, and usually have a longer read range. However, they have a limited life span and are generally more expensive to manufacture.

Passive tags do not need a separate external power source. They derive their operating power from the energy sent by the interrogator. Passive tags are lighter and cheaper than active tags and have virtually unlimited lifetime. Some passive tags contain a battery to maintain internal memory information in read/write applications. The trade-off is that passive tags have a shorter read range than active tags and require a much higher-powered reader to supply the energy needed to operate them.

RF/ID tags can be read-only or read/write. Read-only tags, as the name implies, can only be read, but can be read millions of times. Read/write tags allow the data stored in them to be altered in addition to being read.

Whether the tag is passive or active, read-only or read/write, it requires a receiver circuit. Receiver circuits can be of two types: superheterodyne or crystal video (Fig. 1). Because the superheterodyne receiver contains RF and low noise amplifiers, its detection sensitivity is typically −150 dBm. The crystal video receiver, on the other hand, is limited to only about −55 dBm. However, it is simpler and much cheaper than the superheterodyne receiver, so the RF/ID industry has adopted it for use in tags. The superheterodyne receiver is used in interrogators.

The crystal video receiver of Fig. 1 can take different forms, depending on the application. Four common configurations are shown in Fig. 2. The single-diode circuits offer simplicity and low cost, whereas the voltage doubler circuits provide a higher output for a given input. Each type can be designed with conventional n-type Schottky diodes or zero biased p-type Schottky diodes. If n-type diodes are used, an external dc bias source is needed for detection operation at low input power levels (<1 mW) because of the low saturation current. The p-type zero bias diode does not need a bias source because it has a relatively high saturation current. In addition, it offers the lowest possible cost, size, and complexity, and

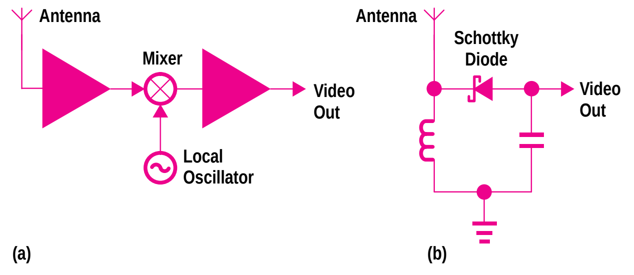


Fig. 1. (a) Superheterodyne receiver. In RF/ID applications, this receiver type is used mainly in interrogators. (b) Crystal video receiver. This type is used in RF/ID tags.

Backscatter RF/ID Systems

Automatic vehicle identification (AVI) is one aspect of intelligent vehicle-highway systems (IVHS). It is a good example of the use of RF/ID technology in a practical application. For example, RFID systems are being integrated into electronic toll collection at bridges so that tolls can be deducted from an account by using information stored in a tag mounted in the vehicle's windshield.

One of the key requirements of these systems is that the stationary reader (interrogator) be able to discriminate between individual tags passing the toll booth without interference from other tags or other transmitters that may be operating at the same frequency. Backscatter modulation technology is one method that can be used for such an application.

A block diagram of a typical transponder (tag) for backscatter technology is shown in Fig. 1. The interrogator (reader) sends a modulated RF signal that is received by the tag. The Schottky diode detector demodulates the signal and transfers the data to the digital circuits of the tag. The radar cross section of the tag is changed by a frequency shift keying encoder and switch driver so that the reflected (backscattered) signal from the tag is modulated and ultimately detected by the reader's receiver antenna. Thus, communication between the tag and reader is established.

By using backscatter technology, interference from nearby transmitters can be avoided, since the reader controls the frequency of operation and can shift it if nearby transmitters are operating at the same frequency. Also, the reflected signal strength from the tag is proportional to the incident interrogator signal, so tags outside the incident beam focus area will reflect a weaker signal that the reader antenna can reject.

Such a backscatter tag can have read/write capabilities that allow flexible digital formats. It can also contain various tag information that can be used in other IVHS applications. A specific minimum field strength is required to put the Schottky detector diode into forward bias so that the tag does not backscatter until the interrogator signal and message data are received, thus minimizing the power requirements of the tag.

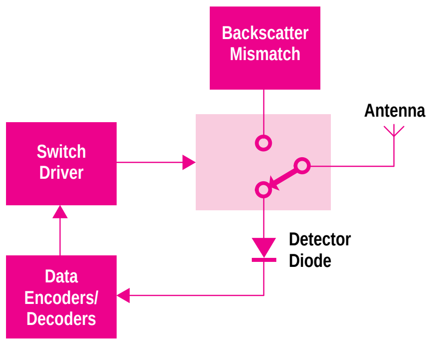


Fig. 1. Typical transponder block diagram for backscatter RF/ID technology.

usually exhibits the lowest flicker noise. It is therefore the diode of choice for RF tag applications.

The performance of an RF/ID system is directly related to the frequency range in which it is used. The higher the frequency, the faster the data transfer rate and the longer the read/write range. The tag's *capture window* is more focused at higher frequency. Metals absorb low-frequency signals more than high-frequency signals, whereas obscuring materials such as dirt and grease absorb high-frequency signals more than low frequency signals. Most RF/ID systems operate in three basic frequency ranges. The high-frequency ranges include 850 to 950 MHz and 2.4 to 2.5 GHz. The low frequency range is 100 to 500 kHz, close to the range of AM radio stations. Some applications, such as auto toll collection, also use 5.86 GHz and 10.5 GHz.

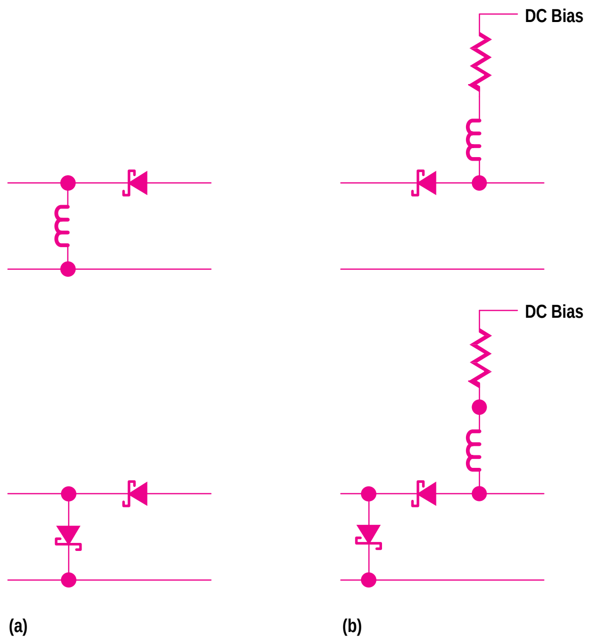


Fig. 2. Different crystal video receiver configurations. (a) Zero bias Schottky diodes. (b) Conventional n-type Schottky diodes.

Device Theory

A Schottky diode is simply a metal layer deposited on a semiconductor such as silicon. To improve its performance and reliability, it can be passivated with silicon dioxide or silicon nitride or both.

The equivalent circuit of a Schottky diode is shown in Fig. 3, along with package parasitic elements. In the diode chip, R_S represents the series resistance of the diode, which includes bulk and contact resistances. Junction capacitance C_j is determined to a first-order approximation by the metal used, the silicon doping, and the active area. R_j is the junction resistance, often called the video resistance R_V, and is a

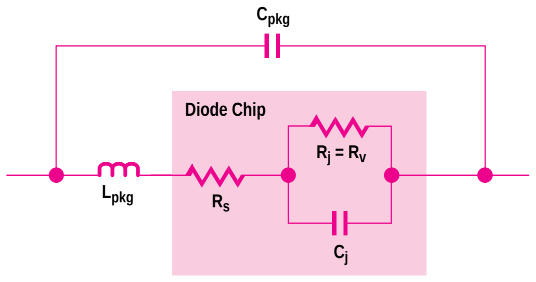


Fig. 3. This model describes an SOT-23 packaged Schottky diode to 10 GHz with good accuracy.

function of the total current flowing through the device. Low C_j, R_v, and R_s are desired for an efficient detector diode.

The total current I flowing through a Schottky diode is given by:

$$I = I_s\left[\exp\left(V_b/nV_t\right) - 1\right] ,$$

where I_s is the diode saturation current, V_b is the voltage across the Schottky barrier, n is the ideality factor, and V_t is the thermal voltage. The voltage across the Schottky barrier is equal to an applied voltage V_a minus any voltage drop across the series resistance R_s, that is, $V_b = V_a - IR_s$. At low bias levels, R_s can be neglected, so $V_b \approx V_a$.

The video resistance is $R_v = dV_b/dI$, so for small I_s:

$$R_v = nV_t/I .$$

For the zero bias condition, $V_a = 0$, at room temperature with n = 1, the video resistance simplifies to:

$$R_v = 0.026/I_s .$$

By increasing I_s, the video resistance of the diode at zero bias is minimized. I_s is increased by proper selection of the metal type and the semiconductor doping. For silicon, p-type generally gives a higher I_s than n-type. However, p-type silicon has higher R_s than n-type silicon with the same doping. Increasing the silicon doping to lower the R_s also increases C_j, which degrades the detector performance. N-type Schottky diodes are seen in mixer applications because of the lower R_s and the fact that R_v can be kept low by using high local oscillator drive levels.

Design Goals

An important performance characteristic used to describe video detector diodes is voltage sensitivity, or γ. This parameter specifies the slope of the curve of output video voltage versus input signal power, that is:

$$V_o = \gamma P_{in} .$$

Neglecting parasitic and reflection losses, voltage sensitivity can be defined as:

$$\gamma = \beta/(\partial I/\partial V) ,$$

where β is the current sensitivity and has a theoretical value[1] of 20 A/W. Using the diode equation (with ideality factor n = 1):

$$\partial I/\partial V = I/0.026 .$$

Therefore,

$$\gamma = 0.52/I .$$

For zero bias detectors, $\gamma = 0.52/I_s$.

This simple analysis of a perfect detector gives a poor approximation to the actual data on existing diodes. To bring the analysis closer to reality, effects of diode junction capacitance, diode series resistance, load resistance, and reflection loss must be considered.

Diode Capacitance and Resistance. In most cases, the junction impedance associated with R_v and C_j is much greater than R_s, especially at low frequencies. However, at high frequencies, the junction impedance is reduced so that the RF power dissipated in R_s is comparable to that of the junction. Incorporating the effects of the diode capacitance and resistance on the current sensitivity,[2] the voltage sensitivity for the zero bias diode becomes:

$$\gamma_1 = 0.52/\left(I_s\left(1 + \omega^2 C_j^2 R_s R_v\right)\right) .$$

where $\omega = 2\pi f$, C_j is junction capacitance, I_s is diode saturation current, R_s is the series resistance, and R_v is the junction resistance.

Load Resistance. The diode resistance R_v at zero bias is usually not small compared to the load resistance R_L. If the diode is considered as a voltage source with impedance R_v feeding the load resistance R_L, the voltage sensitivity will be reduced by the factor $R_L/(R_L + R_v)$, or:

$$\gamma_2 = \gamma_1\left(R_L/\left(R_v + R_L\right)\right) .$$

For example, a typical load resistance is 100 kΩ. If R_v is 5 kΩ, then

$$\gamma_2 = \gamma_1(0.952) .$$

Reflection Loss. Further reduction in voltage sensitivity is caused by reflection losses in the matching circuit in which the diode is used. In Fig. 3, the package capacitance C_{pkg} and package inductance L_{pkg} can be used to determine the packaged diode reflection coefficient. If this diode terminates a 50Ω system, the reflection coefficient ρ is:

$$\rho = \left(Z_D - 50\right)/\left(Z_D + 50\right) ,$$

where Z_D is a function of frequency and the package parasitics. If there is no matching network, the voltage sensitivity can be calculated as:

$$\gamma_3 = \gamma_2\left(1 - \rho^2\right) .$$

The chip, package, and circuit parameters all combine to define an optimum voltage sensitivity for a given application. Our design goal was to develop a diode to operate in the frequency range used in RF/ID tags. Using the voltage sensitivity analysis described above, we hoped to produce an optimum, low-cost, manufacturable part in the shortest time possible.

Implementation and Fabrication

Hewlett-Packard's preeminent zero bias detector diode (HSCH-3486) already provides excellent detection sensitivity in an axially leaded glass package, particularly at high frequencies. To meet our design goals, the project team decided to leverage the HSCH-3486 technology.

We chose the plastic SOT-23 package because of its low manufacturing costs for high-volume products. Using the SOT-23 package, several modifications were possible that we hoped we could take advantage of. Before building prototype devices, we made a detailed device model for the HSCH-3486. The model helped us fabricate an optimum device with minimum design iterations.

Two-dimensional process and device simulators were used to model and predict the performance of the HSCH-3486. Diode parameters such as silicon doping, area, epitaxial layer thickness, metal pad size, and passivation thickness were included to study the effects that these process parameters had on diode electrical performance and ultimately on detector performance. A typical sensitivity analysis (Fig. 4) showed the effect of contact area and epitaxial thickness on voltage sensitivity γ_3, assuming an ideal matching circuit. The model was also used to check for sensitivity to parameters that are not directly measurable during processing, such as surface states and recombination velocities. The model was good for trend analysis but could not be used to predict absolute values until devices were fabricated and tested.

Using the various equations for voltage sensitivity, it is common to plot γ_2 as a function of the saturation current I_s, as shown in Fig. 5, for given values of C_j, R_s, and R_L. Since C_j, R_s, R_v, and I_s interact with one another, it is not simple to lower C_j, say, without increasing R_s. By using the model, we were able to select the best combination of these parameters to maximize the voltage sensitivity at a given frequency. The process model ensured that our design was within the limits of our existing manufacturing capability. In this way, we were able to minimize development costs and time to market.

The fabrication process is relatively simple. Using a heavily doped silicon wafer substrate (to keep R_s low), an epitaxial layer is grown with tight controls on the doping level, thickness, and doping transition width. After silicon dioxide and nitride passivation, photolithography is used to define a contact window. A well-controlled metal process is used to deposit the metal, which defines many of the critical parameters. The metal is etched to an appropriate size for bonding in the plastic package. The wafer is cut into individual die and attached to a leadframe using a silver epoxy. It is then molded into the final plastic configuration. Fig. 6 shows the device cross section and die layout.

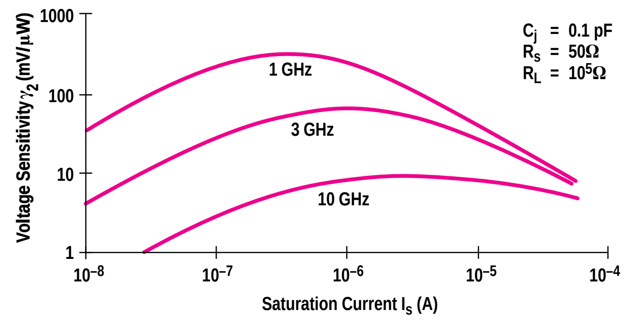


Fig. 5. Voltage sensitivity as a function of saturation current.

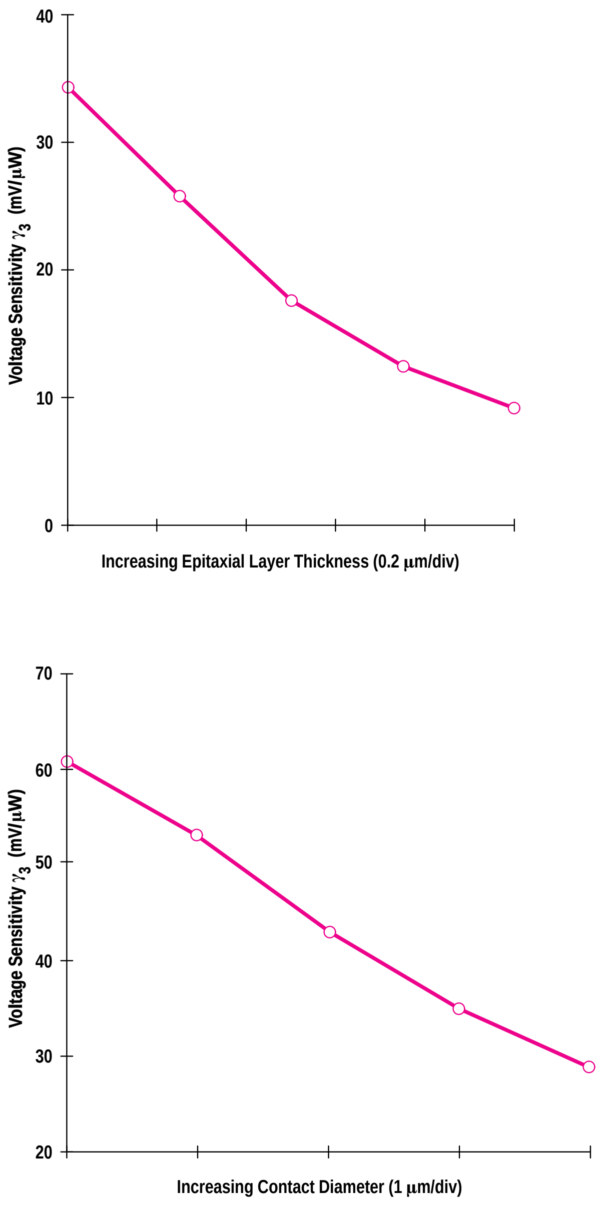


Fig. 4. Effects of epitaxial layer thickness and contact area on voltage sensitivity. f = 5.8 GHz. R_L = 100 kΩ.

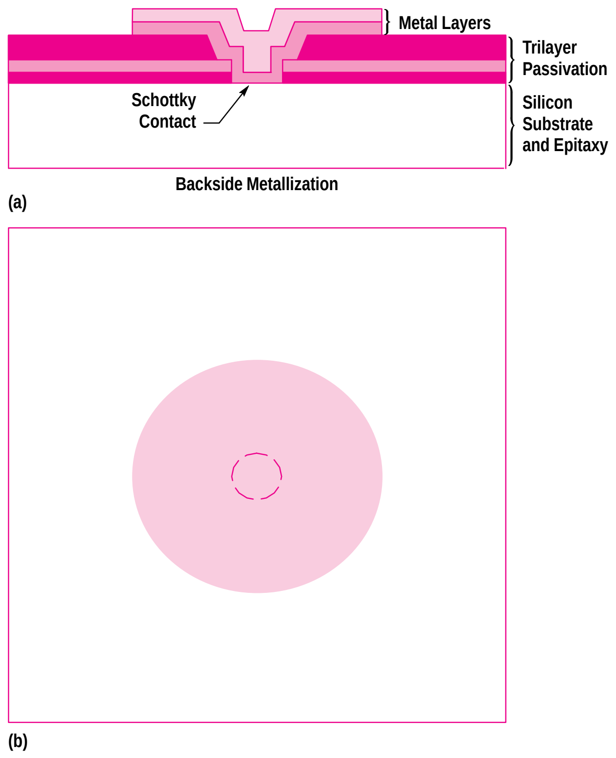


Fig. 6. (a) HSMS-2850 diode cross section. (b) Die layout.

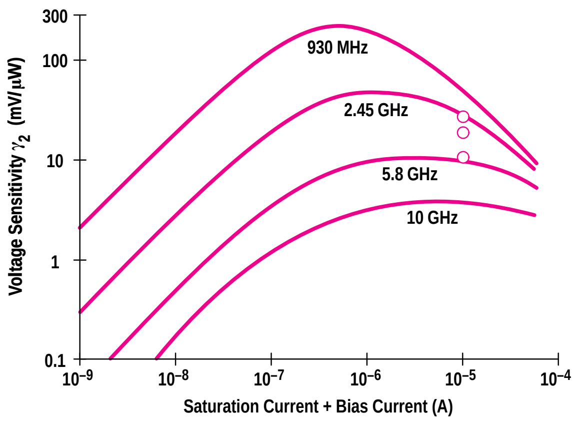


Fig. 7. Calculated voltage sensitivity for zero bias Schottky diodes having $R_s = 55\Omega$ and $C_j = 0.15$ pF. $R_L = 100$ kΩ. Dots show measured values for 930 MHz, 2.45 GHz, and 5.8 GHz.

The packaged device can be 100% tested for various dc parameters such as forward voltage bias V_f and breakdown voltage V_{br}. Many of the dc parameters have been correlated with high-frequency parameters, thus ensuring the performance of each part and eliminating the high costs associated with high-frequency tests.

Performance

The initial lots that were processed after being designed in the process and device simulator performed very closely to the predicted values. Minimal model changes were necessary. In fact, the results were sufficiently good that no design iterations were necessary and the data sheet specifications were set using those lots. Although we did not experience the normal kinds of process variation that we would expect over a long period of time, our confidence in the model accuracy allowed us to simulate these variations to show that the specification would still be met. In addition, we could use the model to determine what process and device parameters could be changed for future improvements to the diode.

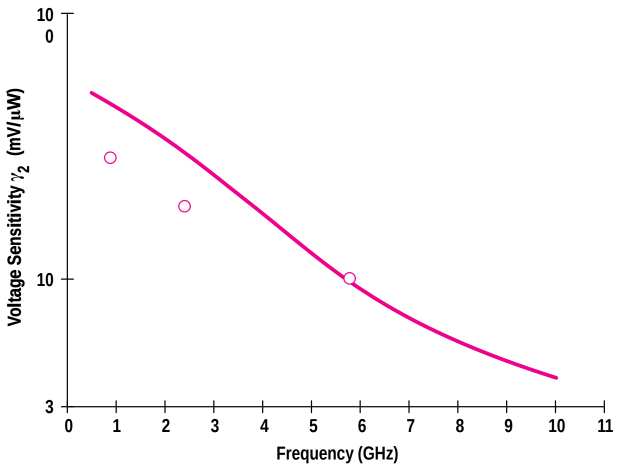


Fig. 8. Calculated voltage sensitivity of the HSMS-2850 zero bias Schottky detector diode. Dots show measured values.

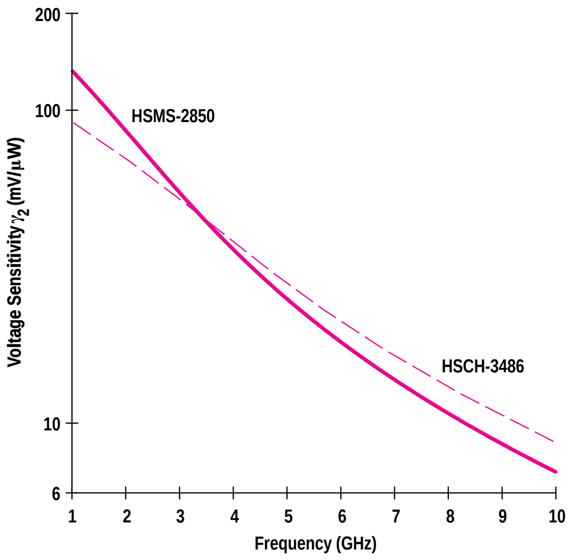


Fig. 9. Comparison of two zero bias Schottky diodes.

Figs. 7 and 8 show actual voltage sensitivity compared to the calculated values.

For comparison with the HSCH-3486 glass package diode, Fig. 9 shows γ_2 as a function of frequency. The different values of C_j, R_s, and I_s of the two diodes cause the HSMS-2850 to provide greater performance at frequencies less than 3 GHz while the HSCH-3486 is superior above 3 GHz. Because of its simpler packaging and testing, the HSMS-2850 is much lower in cost than the HSCH-3486.

Conclusion

Hewlett-Packard's newest silicon zero bias detector diode has one of the best price/performance ratios on the market. We feel that these diodes will become an integral part of many tag applications being designed today and will be considered in future designs and technology. They provide excellent voltage sensitivity for many of the frequency ranges being used in the RF/ID industry at a very low cost.

Acknowledgments

Other members of the project team were Ray Waugh, David Salustri, Bill Lypen, Jatinder Kumar, and Alan McGee. Many other individuals contributed to the successful completion of the project. Special mention should be made of Remedios Solis for wafer process development and testing, Natalia McAfee for packaging in SOT-23, and Vy Frederick for testing the many devices in the SOT-23 package.

References

1. H.A. Watson, *Microwave Semiconductor Devices and Their Circuit Applications*, McGraw-Hill, 1969, p. 379.
2. H.C. Torrey and C.A. Whitmer, *Crystal Rectifiers*, MIT Radiation Laboratory Series, Vol. 15, McGraw-Hill, 1948.