PART EIGHT

Distributed Systems

Traditionally, the data processing function was organized in a centralized fashion. In a centralized data processing architecture, data processing support is provided by one or a cluster of computers, generally large computers, located in a central data processing facility. Many of the tasks performed by such a facility are initiated at the center with the results produced at the center. An example is a payroll application. Other tasks may require interactive access by personnel who are not physically located in the data processing center. For example, a data entry function, such as inventory update, may be performed by personnel at sites throughout the organization. In a centralized architecture, each person is provided with a local terminal that is connected by a communications facility to the central data processing facility.

A fully centralized data processing facility is centralized in many senses of the word:

- **Centralized computers:** One or more computers are located in a central facility. In many cases, there are one or more large mainframe computers, which require special facilities such as air conditioning and a raised floor. In a smaller organization, the central computer or computers are large minicomputers, or midrange systems. The iSeries from IBM is an example of a midrange system.
- **Centralized processing:** All applications are run on the central data processing facility. This includes applications that are clearly central or organization-wide in nature, such as payroll, as well as applications that support the needs of users in a particular organizational unit. As an example of the latter, a product design department may make use of a computer-aided design (CAD) graphics package that runs on the central facility.
- **Centralized data:** All data are stored in files and databases at the central facility and are controlled by and accessible by the central computer or computers. This includes data that are of use to many units in the organization, such as inventory figures, as well as data that support the needs of, and should be used by, only one organizational unit. As an example of the latter, the marketing organization may maintain a database with information derived from customer surveys.

Such a centralized organization has a number of attractive aspects. There may be economies of scale in the purchase and operation of equipment and software. A large central DP shop can afford to have professional programmers on staff to meet the needs of the various departments. Management can maintain control over data processing procurement, enforce standards for programming and data file structure, and design and implement a security policy.

A data processing facility may depart in varying degrees from the centralized data processing organization by implementing a distributed data processing (DDP) strategy. A distributed data processing facility is one in which computers, usually smaller computers, are dispersed throughout an organization. The objective of such dispersion is to process information in a way that is most effective based on operational, economic, and/or geographic considerations, or all three. A DDP facility may include a central facility plus satellite facilities, or it may more nearly resemble a community of peer computing facilities. In either case, some form of interconnection is usually needed; that is, the various computers in the system must be connected to one another. As may be expected, given the characterization of centralized data processing provided here, a DDP facility involves the distribution of computers, processing, and data.

The advantages of DDP include the following

- **Responsiveness:** Local computing facilities can be managed in such a way that they can more directly satisfy the needs of local organizational management than one located in a central facility and intended to satisfy the needs of the total organization.
- **Availability:** With multiple interconnected systems, the loss of any one system should have minimal impact. Key systems and components (e.g., computers with critical applications, printers, mass storage devices) can be replicated so that a backup system can quickly take up the load after a failure.
- **Resource sharing:** Expensive hardware can be shared among users. Data files can be centrally managed and maintained, but with organization-wide access. Staff services, programs, and databases can be developed on an organization-wide basis and distributed to the dispersed facilities.
- **Incremental growth:** In a centralized facility, an increased workload or the need for a new set of applications usually involves a major equipment purchase or a major software upgrade. This involves significant expenditure. In addition, a major change may require conversion or reprogramming of existing applications, with the risk of error and degraded performance. With a distributed system, it is possible to gradually replace applications or systems, avoiding the "all-or-nothing" approach. In addition, old equipment can be left in the facility to run a single application if the cost of moving the application to a new machine is not justified.
- **Increased user involvement and control:** With smaller, more manageable equipment physically located close to the user, the user has greater opportunity to affect system design and operation, either by direction interaction with technical personnel or through the user's immediate superior.
- **End-user productivity:** Distributed systems tend to give more rapid response time to the user, since each piece of equipment is attempting a smaller job. Also, the applications and interfaces of the facility can be optimized to the needs of the organizational unit. Unit managers are in a position to assess the effectiveness of the local portion of the facility and to make the appropriate changes.

To achieve these benefits, the operating system must provide a range of support functions for DDP. These include the software for exchanging data among machines, the capability to cluster machines to achieve high availability and high performance, and the ability to manage processes in a distributed environment.

Note: Chapters 17 and 18 are available online at williamstallings.com/OS/OS6e.html

ROAD MAP FOR PART EIGHT

Chapter 16 Distributed Processing, Client/Server, and Clusters

Chapter 16 looks at the operating system support required for multiple systems to act cooperatively. The chapter looks at the increasingly important concept of client/server computing and the requirements that this architecture places on operating system. The discussion of client/server computing includes a description of two key mechanisms used to implement client/server systems: message passing and remote procedure calls. Chapter 16 also looks at the concept of clusters.

Chapter 17 Networking (Online)

Data network communication and distributed applications rely on underlying communications software that is independent of applications and relieves the application of much of the burden of reliably exchanging data. This communications software is organized into a protocol architecture, the most important incarnation of which is the TCP/IP protocol suite. Chapter 17 introduces the concept of a protocol architecture and provides an overview of TCP/IP.

Chapter 18 Distributed Process Management (Online)

Chapter 18 surveys key issues in developing a distributed operating system. First, we analyze the requirements and mechanisms for process migration, which enables an active process to be moved from one machine to another during the course of its lifetime, to achieve load balancing or availability goals. Then we look at the concept of a distributed global state, which is a vital element in developing a distributed operating system. Finally, we examine concurrency issues related to mutual exclusion and deadlock in a distributed environment.

M16_STAL6329_06_SE_C16.QXD 2/28/08 4:20 AM Page 710

EQA

CHAPTER 16

DISTRIBUTED PROCESSING, CLIENT/SERVER, AND CLUSTERS

In this chapter, we begin with an examination of some of the key concepts in distributed software, including client/server architecture, message passing, and remote procedure calls. Then we examine the increasingly important cluster architecture.

16.1 CLIENT/SERVER COMPUTING

The concept of client/server computing, and related concepts, has become increasingly important in information technology systems. This section begins with a description of the general nature of client/server computing. This is followed by a discussion of alternative ways of organizing the client/server functions. The issue of file cache consistency, raised by the use of file servers, is then examined. Finally, this section introduces the concept of middleware.

What is Client/Server Computing?

As with other new waves in the computer field, client/server computing comes with its own set of jargon words. Table 16.1 lists some of the terms that are commonly found in descriptions of client/server products and applications.

Figure 16.1 attempts to capture the essence of the client/server concept. As the term suggests, a client/server environment is populated by clients and servers. The **client** machines are generally single-user PCs or workstations that provide a highly user-friendly interface to the end user. The client-based station generally presents the type of graphical interface that is most comfortable to users, including the use of windows and a mouse. Microsoft Windows and Macintosh OS provide examples of such interfaces. Client-based applications are tailored for ease of use and include such familiar tools as the spreadsheet.

Each **server** in the client/server environment provides a set of shared services to the clients. The most common type of server currently is the database server, usually controlling a relational database. The server enables many clients to share access to the same database and enables the use of a high-performance computer system to manage the database.

Table 16.1 Client/Server Terminology

Applications Programming Interface (API)
A set of function and call programs that allow clients and servers to intercommunicate

Client
A networked information requester, usually a PC or workstation, that can query database and/or other information from a server

Middleware
A set of drivers, APIs, or other software that improves connectivity between a client application and a server

Relational Database
A database in which information access is limited to the selection of rows that satisfy all search criteria

Server
A computer, usually a high-powered workstation, a minicomputer, or a mainframe, that houses information for manipulation by networked clients

Structured Query Language (SQL)
A language developed by IBM and standardized by ANSI for addressing, creating, updating, or querying relational databases

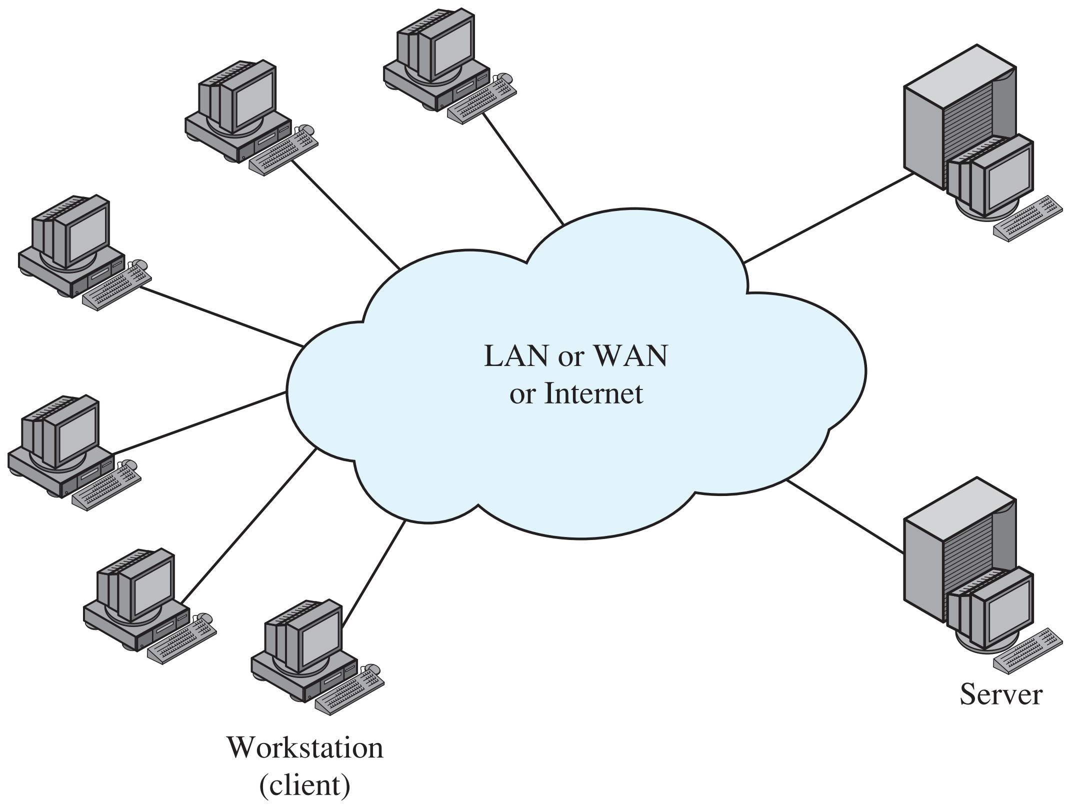


Figure 16.1 **Generic Client/Server Environment**

In addition to clients and servers, the third essential ingredient of the client/server environment is the **network**. Client/server computing is typically distributed computing. Users, applications, and resources are distributed in response to business requirements and linked by a single LAN or WAN or by an internet of networks.

How does a client/server configuration differ from any other distributed processing solution? There are a number of characteristics that stand out and that, together, make client/server distinct from other types of distributed processing:

- There is a heavy reliance on bringing user-friendly applications to the user on his or her own system. This gives the user a great deal of control over the timing and style of computer usage and gives department-level managers the ability to be responsive to their local needs.
- Although applications are dispersed, there is an emphasis on centralizing corporate databases and many network management and utility functions. This enables corporate management to maintain overall control of the total capital investment in computing and information systems and to provide interoperability so that systems are tied together. At the same time it relieves individual departments and divisions of much of the overhead of maintaining

sophisticated computer-based facilities but enables them to choose just about any type of machine and interface they need to access data and information.

- There is a commitment, both by user organizations and vendors, to open and modular systems. This means that the user has more choice in selecting products and in mixing equipment from a number of vendors.
- Networking is fundamental to the operation. Thus, network management and network security have a high priority in organizing and operating information systems.

Client/Server Applications

The key feature of a client/server architecture is the allocation of application-level tasks between clients and servers. Figure 16.2 illustrates the general case. In both client and server, of course, the basic software is an operating system running on the hardware platform. The platforms and the operating systems of client and server may differ. Indeed, there may be a number of different types of client platforms and operating systems and a number of different types of server platforms in a single environment. As long as a particular client and server share the same communications protocols and support the same applications, these lower-level differences are irrelevant.

It is the communications software that enables client and server to interoperate. The principal example of such software is TCP/IP. Of course, the point of all of this support software (communications and operating system) is to provide a base for distributed applications. Ideally, the actual functions performed by the application can be split up between client and server in a way that optimizes the use of resources. In some cases, depending on the application needs, the bulk of the applications software executes at the server, while in other cases, most of the application logic is located at the client.

An essential factor in the success of a client/server environment is the way in which the user interacts with the system as a whole. Thus, the design of the user

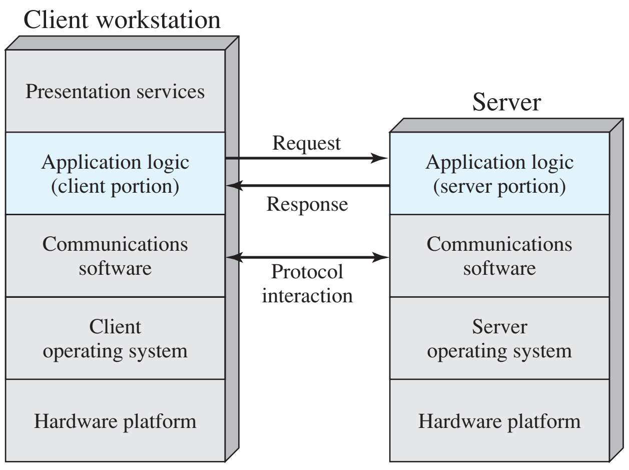


Figure 16.2 Generic Client/Server Architecture

interface on the client machine is critical. In most client/server systems, there is heavy emphasis on providing a **graphical user interface (GUI)** that is easy to use, easy to learn, yet powerful and flexible. Thus, we can think of a presentation services module in the client workstation that is responsible for providing a user-friendly interface to the distributed applications available in the environment.

Database Applications As an example that illustrates the concept of splitting application logic between client and server, let us consider one of the most common families of client/server applications: those that use relational databases. In this environment, the server is essentially a database server. Interaction between client and server is in the form of transactions in which the client makes a database request and receives a database response.

Figure 16.3 illustrates, in general terms, the architecture of such a system. The server is responsible for maintaining the database, for which purpose a complex database management system software module is required. A variety of different applications that make use of the database can be housed on client machines. The "glue" that ties client and server together is software that enables the client to make requests for access to the server's database. A popular example of such logic is the structured query language (SQL).

Figure 16.3 suggests that all of the application logic—the software for "number crunching" or other types of data analysis—is on the client side, while the server is only concerned with managing the database. Whether such a configuration is appropriate depends on the style and intent of the application. For example, suppose that the primary purpose is to provide online access for record lookup. Figure 16.4a suggests how this might work. Suppose that the server is maintaining a database of 1 million records (called rows in relational database terminology), and the user wants to perform a lookup that should result in zero, one, or at most a few records.

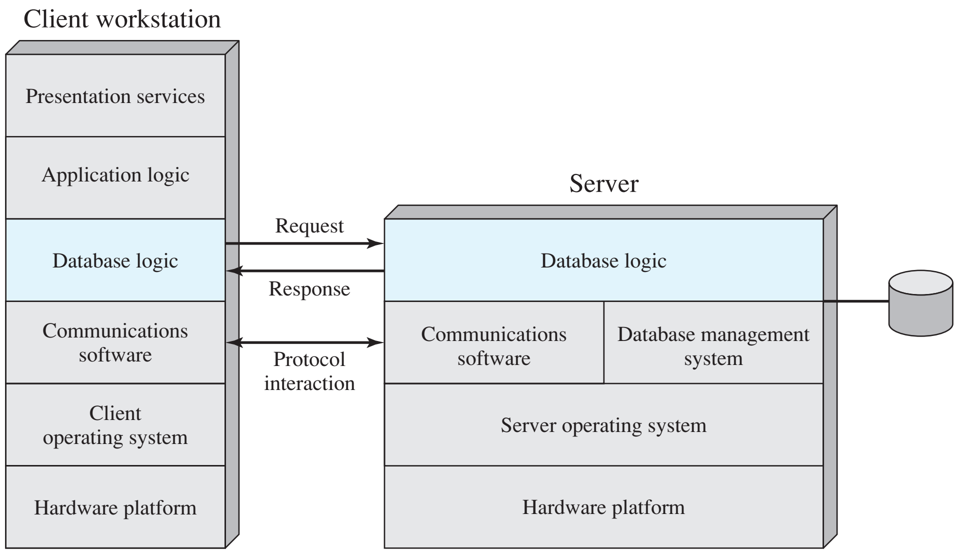


Figure 16.3 **Client/Server Architecture for Database Applications**

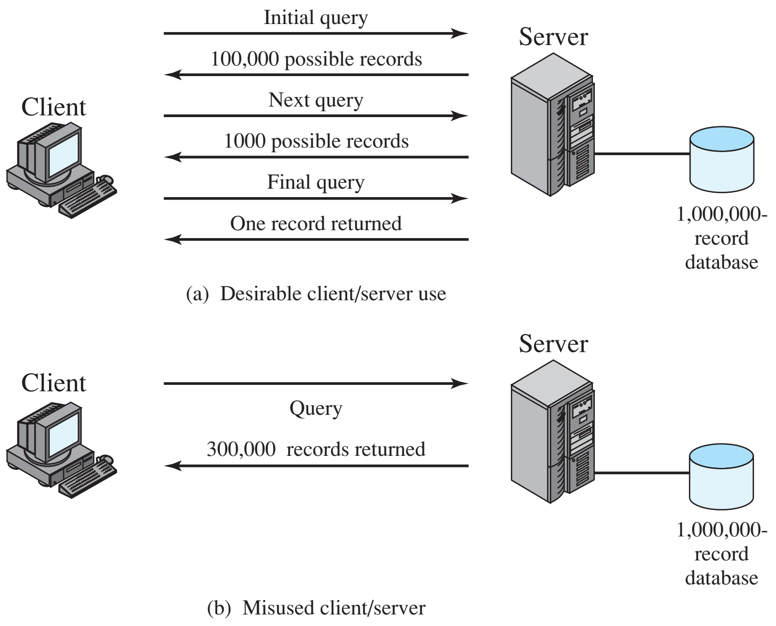


Figure 16.4 **Client/Server Database Usage**

The user could search for these records using a number of search criteria (e.g., records older than 1992; records referring to individuals in Ohio; records referring to a specific event or characteristic, etc.). An initial client query may yield a server response that there are 100,000 records that satisfy the search criteria. The user then adds additional qualifiers and issues a new query. This time, a response indicating that there are 1000 possible records is returned. Finally, the client issues a third request with additional qualifiers. The resulting search criteria yield a single match, and the record is returned to the client.

The preceding application is well suited to a client/server architecture for two reasons:

1. There is a massive job of sorting and searching the database. This requires a large disk or bank of disks, a high-speed CPU, and a high-speed I/O architecture. Such capacity and power is not needed and is too expensive for a single-user workstation or PC.
2. It would place too great a traffic burden on the network to move the entire 1-million-record file to the client for searching. Therefore, it is not enough for the server just to be able to retrieve records on behalf of a client; the server needs to have database logic that enables it to perform searches on behalf of a client.

Now consider the scenario of Figure 16.4b, which has the same 1-million-record database. In this case, a single query results in the transmission of 300,000 records over the network. This might happen if, for example, the user wishes to find the grand total or mean value of some field across many records or even the entire database.

Clearly, this latter scenario is unacceptable. One solution to this problem, which maintains the client/server architecture with all of its benefits, is to move part of the application logic over to the server. That is, the server can be equipped with application logic for performing data analysis as well as data retrieval and data searching.

Classes of Client/Server Applications Within the general framework of client/server, there is a spectrum of implementations that divide the work between client and server differently. Figure 16.5 illustrates in general terms some of the major options for database applications. Other splits are possible, and the options may have a different characterization for other types of applications. In any case, it is useful to examine this figure to get a feel for the kind of tradeoffs possible.

Figure 16.5 depicts four classes:

- **Host-based processing:** Host-based processing is not true client/server computing as the term is generally used. Rather, host-based processing refers to the traditional mainframe environment in which all or virtually all of the processing is done on a central host. Often the user interface is via a dumb terminal. Even if the user is employing a microcomputer, the user's station is generally limited to the role of a terminal emulator.

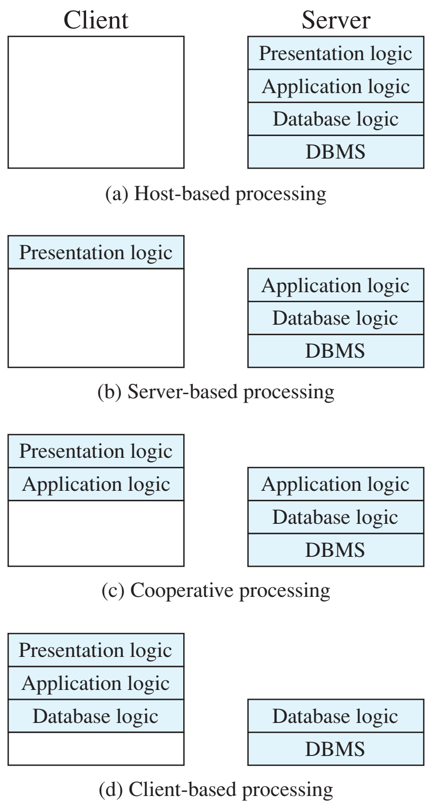


Figure 16.5 Classes of Client/Server Applications

- **Server-based processing:** The most basic class of client/server configuration is one in which the client is principally responsible for providing a graphical user interface, while virtually all of the processing is done on the server. This configuration is typical of early client/server efforts, especially departmental-level systems. The rationale behind such configurations is that the user workstation is best suited to providing a user-friendly interface and that databases and applications can easily be maintained on central systems. Although the user gains the advantage of a better interface, this type of configuration does not generally lend itself to any significant gains in productivity or to any fundamental changes in the actual business functions that the system supports.
- **Client-based processing:** At the other extreme, virtually all application processing may be done at the client, with the exception of data validation routines and other database logic functions that are best performed at the server. Generally, some of the more sophisticated database logic functions are housed on the client side. This architecture is perhaps the most common client/server approach in current use. It enables the user to employ applications tailored to local needs.
- **Cooperative processing:** In a cooperative processing configuration, the application processing is performed in an optimized fashion, taking advantage of the strengths of both client and server machines and of the distribution of data. Such a configuration is more complex to set up and maintain but, in the long run, this type of configuration may offer greater user productivity gains and greater network efficiency than other client/server approaches.

Figures 16.5c and d correspond to configurations in which a considerable fraction of the load is on the client. This so-called **fat client** model has been popularized by application development tools such as Sybase Inc.'s PowerBuilder and Gupta Corp.'s SQL Windows. Applications developed with these tools are typically departmental in scope, supporting between 25 and 150 users [ECKE95]. The main benefit of the fat client model is that it takes advantage of desktop power, offloading application processing from servers and making them more efficient and less likely to be bottlenecks.

There are, however, several disadvantages to the fat client strategy. The addition of more functions rapidly overloads the capacity of desktop machines, forcing companies to upgrade. If the model extends beyond the department to incorporate many users, the company must install high-capacity LANs to support the large volumes of transmission between the thin servers and the fat clients. Finally, it is difficult to maintain, upgrade, or replace applications distributed across tens or hundreds of desktops.

Figure 16.5b is representative of a **thin client** approach. This approach more nearly mimics the traditional host-centered approach and is often the migration path for evolving corporate-wide applications from the mainframe to a distributed environment.

Three-Tier Client/Server Architecture

The traditional client/server architecture involves two levels, or tiers: a client tier and a server tier. A three-tier architecture is also common (Figure 16.6). In this architecture, the application software is distributed among three types of machines: a user machine, a middle-tier server, and a backend server. The user machine is the client machine we have been

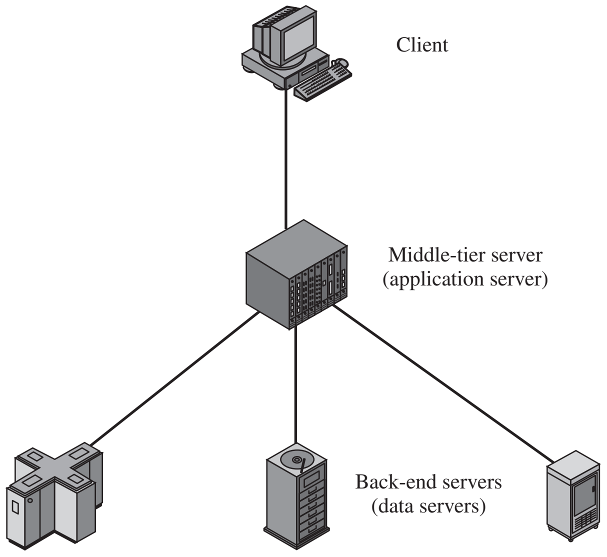


Figure 16.6 **Three-tier Client/Server Architecture**

discussing and, in the three-tier model, is typically a thin client. The middle-tier machines are essentially gateways between the thin user clients and a variety of backend database servers. The middle-tier machines can convert protocols and map from one type of database query to another. In addition, the middle-tier machine can merge/integrate results from different data sources. Finally, the middle-tier machine can serve as a gateway between the desktop applications and the backend legacy applications by mediating between the two worlds.

The interaction between the middle-tier server and the backend server also follows the client/server model. Thus, the middle-tier system acts as both a client and a server.

File Cache Consistency When a file server is used, performance of file I/O can be noticeably degraded relative to local file access because of the delays imposed by the network. To reduce this performance penalty, individual systems can use file caches to hold recently accessed file records. Because of the principle of locality, use of a local file cache should reduce the number of remote server accesses that must be made.

Figure 16.7 illustrates a typical distributed mechanism for caching files among a networked collection of workstations. When a process makes a file access, the request is presented first to the cache of the process's workstation ("file traffic"). If not satisfied there, the request is passed either to the local disk, if the file is stored there ("disk traffic"), or to a file server, where the file is stored ("server traffic"). At the server, the server's cache is first interrogated and, if there is a miss, then the server's disk is accessed. The dual caching approach is used to reduce communications traffic (client cache) and disk I/O (server cache).

When caches always contain exact copies of remote data, we say that the caches are **consistent**. It is possible for caches to become inconsistent when the remote data are changed and the corresponding obsolete local cache copies are not discarded. This can happen if one client modifies a file that is also cached by other

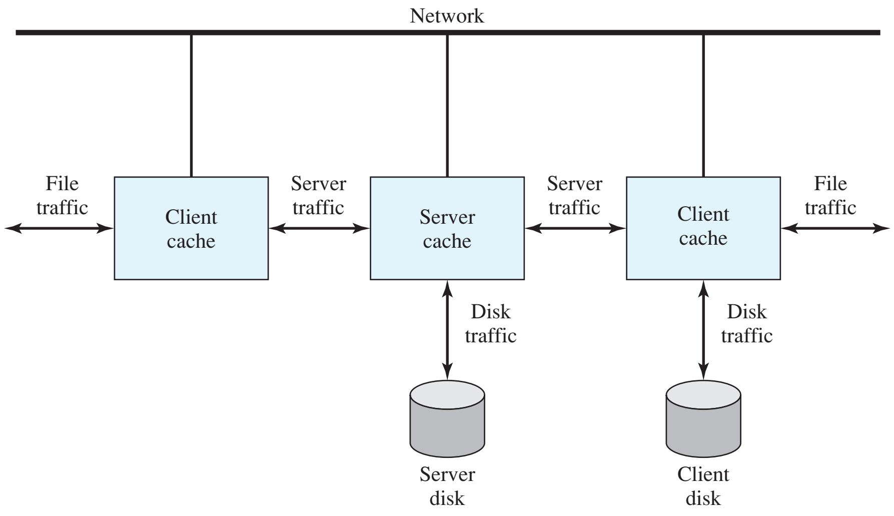


Figure 16.7 Distributied File Cacheing in Sprite

clients. The difficulty is actually at two levels. If a client adopts a policy of immediately writing any changes to a file back to the server, then any other client that has a cache copy of the relevant portion of the file will have obsolete data. The problem is made even worse if the client delays writing back changes to the server. In that case, the server itself has an obsolete version of the file, and new file read requests to the server might obtain obsolete data. The problem of keeping local cache copies up to date to changes in remote data is known as the **cache consistency** problem.

The simplest approach to cache consistency is to use file-locking techniques to prevent simultaneous access to a file by more than one client. This guarantees consistency at the expense of performance and flexibility. A more powerful approach is provided with the facility in Sprite [NELS88, OUST88]. Any number of remote processes may open a file for read and create their own client cache. But when an open file request to a server requests write access and other processes have the file open for read access, the server takes two actions. First, it notifies the writing process that, although it may maintain a cache, it must write back all altered blocks immediately upon update. There can be at most one such client. Second, the server notifies all reading processes that have the file open that the file is no longer cacheable.

Middleware

The development and deployment of client/server products has far outstripped efforts to standardize all aspects of distributed computing, from the physical layer up to the application layer. This lack of standards makes it difficult to implement an integrated, multivendor, enterprise-wide client/server configuration. Because much of the benefit of the client/server approach is tied up with its modularity and the ability to mix and match platforms and applications to provide a business solution, this interoperability problem must be solved.

To achieve the true benefits of the client/server approach, developers must have a set of tools that provide a uniform means and style of access to system resources across all platforms. This will enable programmers to build applications that not only look and feel the same on various PCs and workstations but that use the same method to access data regardless of the location of that data.

The most common way to meet this requirement is by the use of standard programming interfaces and protocols that sit between the application above and communications software and operating system below. Such standardized interfaces and protocols have come to be referred to as middleware. With standard programming interfaces, it is easy to implement the same application on a variety of server types and workstation types. This obviously benefits the customer, but vendors are also motivated to provide such interfaces. The reason is that customers buy applications, not servers; customers will only choose among those server products that run the applications they want. The standardized protocols are needed to link these various server interfaces back to the clients that need access to them.

There is a variety of middleware packages ranging from the very simple to the very complex. What they all have in common is the capability to hide the complexities and disparities of different network protocols and operating systems. Client and server vendors generally provide a number of the more popular middleware packages as options. Thus, a user can settle on a particular middleware strategy and then assemble equipment from various vendors that support that strategy.

Middleware Architecture Figure 16.8 suggests the role of middleware in a client/server architecture. The exact role of the middleware component will depend on the style of client/server computing being used. Referring back to Figure 16.5, recall that there are a number of different client/server approaches, depending on the way in which application functions are split up. In any case, Figure 16.8 gives a good general idea of the architecture involved.

Note that there is both a client and server component of middleware. The basic purpose of middleware is to enable an application or user at a client to access

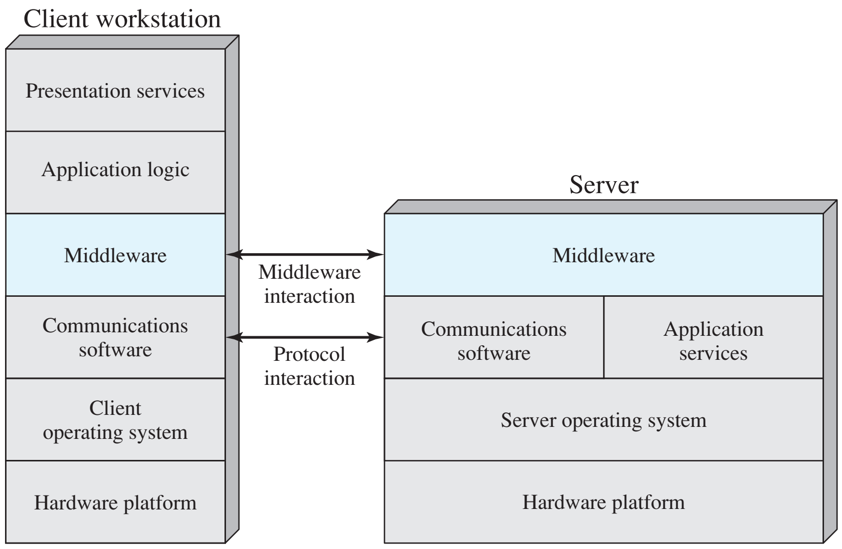


Figure 16.8 The Role of Middleware in Client/Server Architecture

a variety of services on servers without being concerned about differences among servers. To look at one specific application area, the structured query language (SQL) is supposed to provide a standardized means for access to a relational database by either a local or remote user or application. However, many relational database vendors, although they support SQL, have added their own proprietary extensions to SQL. This enables vendors to differentiate their products but also creates potential incompatibilities.

As an example, consider a distributed system used to support, among other things, the personnel department. The basic employee data, such as employee name and address, might be stored on a Gupta database, whereas salary information might be contained on an Oracle database. When a user in the personnel department requires access to particular records, that user does not want to be concerned with which vendor's database contains the records needed. Middleware provides a layer of software that enables uniform access to these differing systems.

It is instructive to look at the role of middleware from a logical, rather than an implementation, point of view. This viewpoint is illustrated in Figure 16.9. Middleware enables the realization of the promise of distributed client/server computing. The entire distributed system can be viewed as a set of applications and resources available to users. Users need not be concerned with the location of data or indeed the location of applications. All applications operate over a uniform applications programming interface (API). The middleware, which cuts across all client and server platforms, is responsible for routing client requests to the appropriate server.

Although there is a wide variety of middleware products, these products are typically based on one of two underlying mechanisms: message passing or remote procedure calls. These two methods are examined in the next two sections.

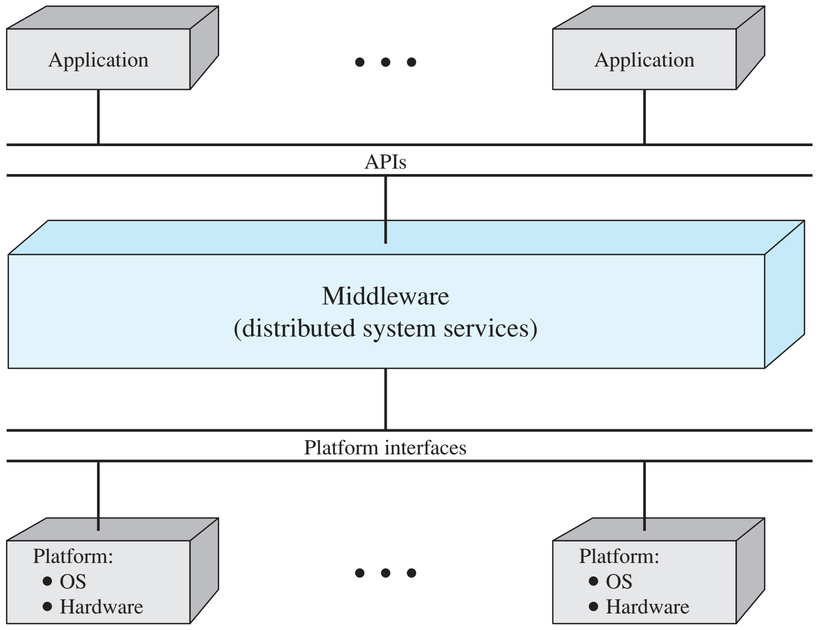


Figure 16.9 Logical View of Middleware

16.2 DISTRIBUTED MESSAGE PASSING

It is usually the case in a distributed processing systems that the computers do not share main memory; each is an isolated computer system. Thus, interprocessor communication techniques that rely on shared memory, such as semaphores, cannot be used. Instead, techniques that rely on message passing are used. In this section and the next, we look at the two most common approaches. The first is the straightforward application of messages as they are used in a single system. The second is a separate technique that relies on message passing as a basic function: the remote procedure call.

Figure 16.10a shows the use of message passing to implement client/server functionality. A client process requires some service (e.g., read a file, print) and

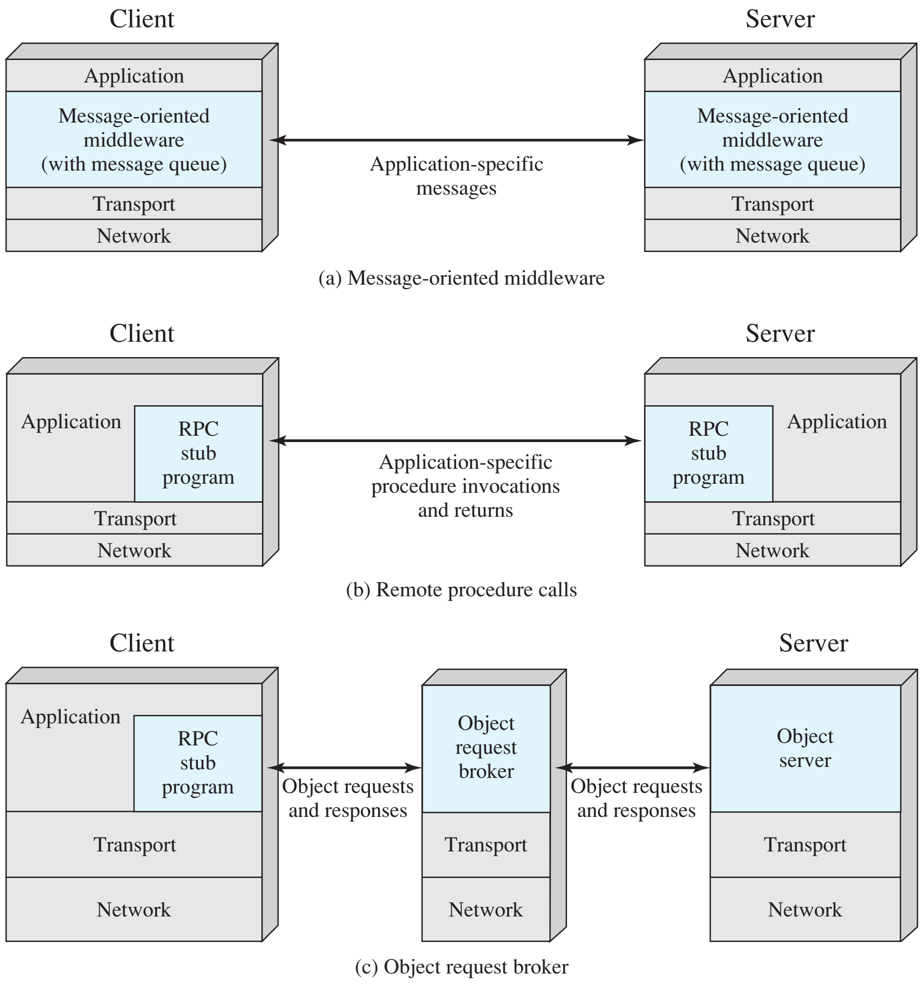

Client
Server
Application
Message-oriented middleware (with message queue)
Transport
Network
Application-specific messages
(a) Message-oriented middleware
Application
RPC stub program
Application-specific procedure invocations and returns
(b) Remote procedure calls
Object requests and responses
Object request broker
Object server
(c) Object request broker

Figure 16.10 **Middleware Mechanisms**

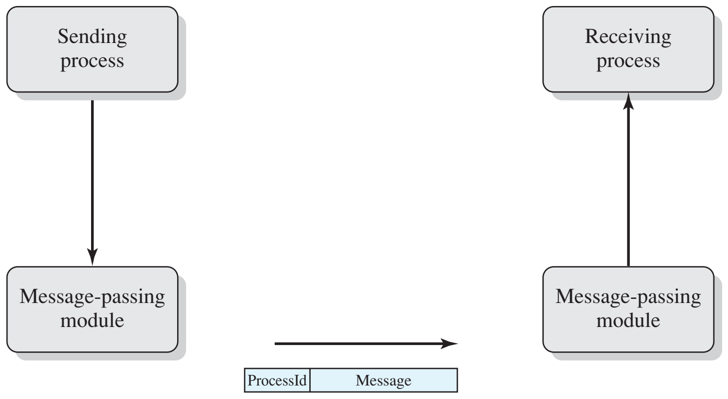


Figure 16.11 **Basic Message-Passing Primitives**

sends a message containing a request for service to a server process. The server process honors the request and sends a message containing a reply. In its simplest form, only two functions are needed: Send and Receive. The Send function specifies a destination and includes the message content. The Receive function tells from whom a message is desired (including "all") and provides a buffer where the incoming message is to be stored.

Figure 16.11 suggests an implementation for message passing. Processes make use of the services of a message-passing module. Service requests can be expressed in terms of primitives and parameters. A primitive specifies the function to be performed, and the parameters are used to pass data and control information. The actual form of a primitive depends on the message-passing software. It may be a procedure call or it may itself be a message to a process that is part of the operating system.

The Send primitive is used by the process that desires to send the message. Its parameters are the identifier of the destination process and the contents of the message. The message-passing module constructs a data unit that includes these two elements. This data unit is sent to the machine that hosts the destination process, using some sort of communications facility, such as TCP/IP. When the data unit is received in the target system, it is routed by the communications facility to the message-passing module. This module examines the process ID field and stores the message in the buffer for that process.

In this scenario, the receiving process must announce its willingness to receive messages by designating a buffer area and informing the message-passing module by a Receive primitive. An alternative approach does not require such an announcement. Instead, when the message-passing module receives a message, it signals the destination process with some sort of Receive signal and then makes the received message available in a shared buffer.

Several design issues are associated with distributed message passing, and these are addressed in the remainder of this section.

Reliability versus Unreliability

A reliable message-passing facility is one that guarantees delivery if possible. Such a facility makes use of a reliable transport protocol or similar logic and performs error checking, acknowledgment, retransmission, and reordering of misordered messages. Because delivery is guaranteed, it is not necessary to let the sending process know that the message was delivered. However, it might be useful to provide an acknowledgment back to the sending process so that it knows that delivery has already taken place. In either case, if the facility fails to achieve delivery (e.g., persistent network failure, crash of destination system), the sending process is notified of the failure.

At the other extreme, the message-passing facility may simply send the message out into the communications network but will report neither success nor failure. This alternative greatly reduces the complexity and processing and communications overhead of the message-passing facility. For those applications that require confirmation that a message has been delivered, the applications themselves may use request and reply messages to satisfy the requirement.

Blocking versus Nonblocking

With nonblocking, or asynchronous, primitives, a process is not suspended as a result of issuing a Send or Receive. Thus, when a process issues a Send primitive, the operating system returns control to the process as soon as the message has been queued for transmission or a copy has been made. If no copy is made, any changes made to the message by the sending process before or even while it is being transmitted are made at the risk of the process. When the message has been transmitted or copied to a safe place for subsequent transmission, the sending process is interrupted to be informed that the message buffer may be reused. Similarly, a nonblocking Receive is issued by a process that then proceeds to run. When a message arrives, the process is informed by interrupt, or it can poll for status periodically.

Nonblocking primitives provide for efficient, flexible use of the message-passing facility by processes. The disadvantage of this approach is that it is difficult to test and debug programs that use these primitives. Irreproducible, timing-dependent sequences can create subtle and difficult problems.

The alternative is to use blocking, or synchronous, primitives. A blocking Send does not return control to the sending process until the message has been transmitted (unreliable service) or until the message has been sent and an acknowledgment received (reliable service). A blocking Receive does not return control until a message has been placed in the allocated buffer.

16.3 REMOTE PROCEDURE CALLS

A variation on the basic message-passing model is the remote procedure call. This is now a widely accepted and common method for encapsulating communication in a distributed system. The essence of the technique is to allow programs on different machines to interact using simple procedure call/return semantics, just as if the two programs were on the same machine. That is, the procedure call is used for access to remote services. The popularity of this approach is due to the following advantages.

1. The procedure call is a widely accepted, used, and understood abstraction.
2. The use of remote procedure calls enables remote interfaces to be specified as a set of named operations with designated types. Thus, the interface can

be clearly documented and distributed programs can be statically checked for type errors.

3. Because a standardized and precisely defined interface is specified, the communication code for an application can be generated automatically.
4. Because a standardized and precisely defined interface is specified, developers can write client and server modules that can be moved among computers and operating systems with little modification and recoding.

The remote procedure call mechanism can be viewed as a refinement of reliable, blocking message passing. Figure 16.10b illustrates the general architecture, and Figure 16.12 provides a more detailed look. The calling program makes a normal procedure call with parameters on its machine. For example,

$$\text{CALL P}(X,Y)$$

where

P = procedure name
X = passed arguments
Y = returned values

It may or may not be transparent to the user that the intention is to invoke a remote procedure on some other machine. A dummy or stub procedure P must be included in the caller's address space or be dynamically linked to it at call time. This procedure creates a message that identifies the procedure being called and includes the parameters. It then sends this message to a remote system and waits for a reply. When a reply is received, the stub procedure returns to the calling program, providing the returned values.

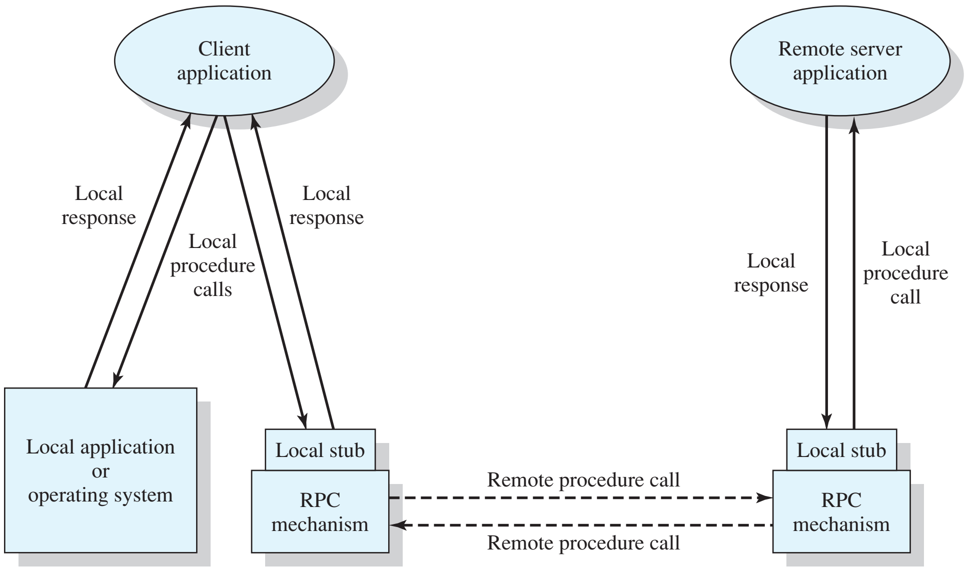


Figure 16.12 **Remote Procedure Call Mechanism**

At the remote machine, another stub program is associated with the called procedure. When a message comes in, it is examined and a local CALL P(X, Y) is generated. This remote procedure is thus called locally, so its normal assumptions about where to find parameters, the state of the stack, and so on are identical to the case of a purely local procedure call.

Several design issues are associated with remote procedure calls, and these are addressed in the remainder of this section.

Parameter Passing

Most programming languages allow parameters to be passed as values (call by value) or as pointers to a location that contains the value (call by reference). Call by value is simple for a remote procedure call: the parameters are simply copied into the message and sent to the remote system. It is more difficult to implement call by reference. A unique, systemwide pointer is needed for each object. The overhead for this capability may not be worth the effort.

Parameter Representation

Another issue is how to represent parameters and results in messages. If the called and calling programs are in identical programming languages on the same type of machines with the same operating system, then the representation requirement may present no problems. If there are differences in these areas, then there will probably be differences in the ways in which numbers and even text are represented. If a full-blown communications architecture is used, then this issue is handled by the presentation layer. However, the overhead of such an architecture has led to the design of remote procedure call facilities that bypass most of the communications architecture and provide their own basic communications facility. In that case, the conversion responsibility falls on the remote procedure call facility (e.g., see [GIBB87]).

The best approach to this problem is to provide a standardized format for common objects, such as integers, floating-point numbers, characters, and character strings. Then the native parameters on any machine can be converted to and from the standardized representation.

Client/Server Binding

Binding specifies how the relationship between a remote procedure and the calling program will be established. A binding is formed when two applications have made a logical connection and are prepared to exchange commands and data.

Nonpersistent binding means that a logical connection is established between the two processes at the time of the remote procedure call and that as soon as the values are returned, the connection is dismantled. Because a connection requires the maintenance of state information on both ends, it consumes resources. The nonpersistent style is used to conserve those resources. On the other hand, the overhead involved in establishing connections makes nonpersistent binding inappropriate for remote procedures that are called frequently by the same caller.

With **persistent binding**, a connection that is set up for a remote procedure call is sustained after the procedure return. The connection can then be used for future remote procedure calls. If a specified period of time passes with no activity on the connection, then the connection is terminated. For applications that make many repeated calls to remote procedures, persistent binding maintains the logical connection and allows a sequence of calls and returns to use the same connection.

Synchronous versus Asynchronous

The concepts of synchronous and asynchronous remote procedure calls are analogous to the concepts of blocking and nonblocking messages. The traditional remote procedure call is synchronous, which requires that the calling process wait until the called process returns a value. Thus, the **synchronous RPC** behaves much like a subroutine call.

The synchronous RPC is easy to understand and program because its behavior is predictable. However, it fails to exploit fully the parallelism inherent in distributed applications. This limits the kind of interaction the distributed application can have, resulting in lower performance.

To provide greater flexibility, various **asynchronous RPC** facilities have been implemented to achieve a greater degree of parallelism while retaining the familiarity and simplicity of the RPC [ANAN92]. Asynchronous RPCs do not block the caller; the replies can be received as and when they are needed, thus allowing client execution to proceed locally in parallel with the server invocation.

A typical asynchronous RPC use is to enable a client to invoke a server repeatedly so that the client has a number of requests in the pipeline at one time, each with its own set of data. Synchronization of client and server can be achieved in one of two ways:

1. A higher-layer application in the client and server can initiate the exchange and then check at the end that all requested actions have been performed.
2. A client can issue a string of asynchronous RPCs followed by a final synchronous RPC. The server will respond to the synchronous RPC only after completing all of the work requested in the preceding asynchronous RPCs.

In some schemes, asynchronous RPCs require no reply from the server and the server cannot send a reply message. Other schemes either require or allow a reply, but the caller does not wait for the reply.

Object-Oriented Mechanisms

As object-oriented technology becomes more prevalent in operating system design, client/server designers have begun to embrace this approach. In this approach, clients and servers ship messages back and forth between objects. Object communications may rely on an underlying message or RPC structure or be developed directly on top of object-oriented capabilities in the operating system.

A client that needs a service sends a request to an object request broker, which acts as a directory of all the remote service available on the network (Figure 16.10c). The broker calls the appropriate object and passes along any relevant data. Then the remote object services the request and replies to the broker, which returns the response to the client.

The success of the object-oriented approach depends on standardization of the object mechanism. Unfortunately, there are several competing designs in this area. One is Microsoft's Component Object Model (COM), the basis for Object Linking and Embedding (OLE). A competing approach, developed by the Object Management Group, is the Common Object Request Broker Architecture (CORBA), which has wide industry support. IBM, Apple, Sun, and many other vendors support the CORBA approach.

16.4 CLUSTERS

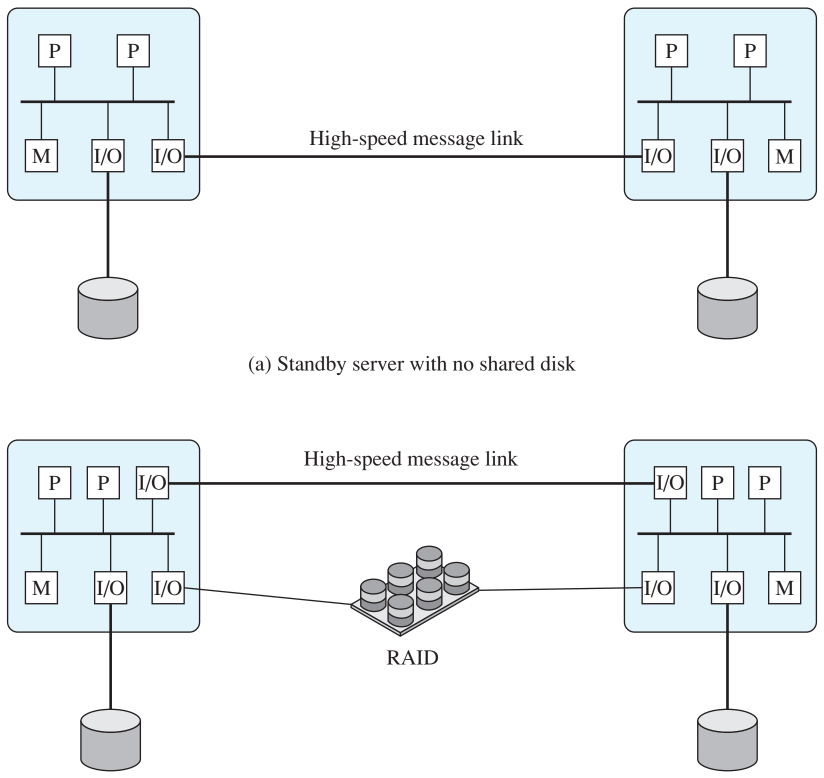
An important and relatively recent development computer system design is clustering. Clustering is an alternative to symmetric multiprocessing (SMP) as an approach to providing high performance and high availability and is particularly attractive for server applications. We can define a cluster as a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster; in the literature, each computer in a cluster is typically referred to as a *node*.

[BREW97] lists four benefits that can be achieved with clustering. These can also be thought of as objectives or design requirements:

- **Absolute scalability:** It is possible to create large clusters that far surpass the power of even the largest standalone machines. A cluster can have dozens or even hundreds of machines, each of which is a multiprocessor.
- **Incremental scalability:** A cluster is configured in such a way that it is possible to add new systems to the cluster in small increments. Thus, a user can start out with a modest system and expand it as needs grow, without having to go through a major upgrade in which an existing small system is replaced with a larger system.
- **High availability:** Because each node in a cluster is a standalone computer, the failure of one node does not mean loss of service. In many products, fault tolerance is handled automatically in software.
- **Superior price/performance:** By using commodity building blocks, it is possible to put together a cluster with equal or greater computing power than a single large machine, at much lower cost.

Cluster Configurations

In the literature, clusters are classified in a number of different ways. Perhaps the simplest classification is based on whether the computers in a cluster share access to the same disks. Figure 16.13a shows a two-node cluster in which the only interconnection is by means of a high-speed link that can be used for message exchange to coordinate cluster activity. The link can be a LAN that is shared with other computers that are not part of the cluster or the link can be a dedicated interconnection facility. In the latter case, one or more of the computers in the cluster will have a link to a LAN or WAN so that there is a connection between the server cluster and



(a) Standby server with no shared disk

(b) Shared disk

Figure 16.13 Cluster Configurations

remote client systems. Note that in the figure, each computer is depicted as being a multiprocessor. This is not necessary but does enhance both performance and availability.

In the simple classification depicted in Figure 16.13, the other alternative is a shared-disk cluster. In this case, there generally is still a message link between nodes. In addition, there is a disk subsystem that is directly linked to multiple computers within the cluster. In Figure 16.13b, the common disk subsystem is a RAID system. The use of RAID or some similar redundant disk technology is common in clusters so that the high availability achieved by the presence of multiple computers is not compromised by a shared disk that is a single point of failure.

A clearer picture of the range of clustering approaches can be gained by looking at functional alternatives. A white paper from Hewlett Packard [HP96] provides a useful classification along functional lines (Table 16.2), which we now discuss.

A common, older method, known as **passive standby**, is simply to have one computer handle all of the processing load while the other computer remains inactive, standing by to take over in the event of a failure of the primary. To coordinate the machines, the active, or primary, system periodically sends a "heartbeat" message to the standby machine. Should these messages stop arriving, the standby assumes that the primary server has failed and puts itself into operation. This approach increases

Table 16.2 Clustering Methods: Benefits and Limitations

Clustering Method	Description	Benefits	Limitations
Passive Standby	A secondary server takes over in case of primary server failure.	Easy to implement.	High cost because the secondary server is unavailable for other processing tasks.
Active Secondary	The secondary server is also used for processing tasks.	Reduced cost because secondary servers can be used for processing.	Increased complexity.
Separate Servers	Separate servers have their own disks. Data is continuously copied from primary to secondary server.	High availability.	High network and server overhead due to copying operations.
Servers Connected to Disks	Servers are cabled to the same disks, but each server owns its disks. If one server fails, its disks are taken over by the other server.	Reduced network and server overhead due to elimination of copying operations.	Usually requires disk mirroring or RAID technology to compensate for risk of disk failure.
Servers Share Disks	Multiple servers simultaneously share access to disks.	Low network and server overhead. Reduced risk of downtime caused by disk failure.	Requires lock manager software. Usually used with disk mirroring or RAID technology.

availability but does not improve performance. Further, if the only information that is exchanged between the two systems is a heartbeat message, and if the two systems do not share common disks, then the standby provides a functional backup but has no access to the databases managed by the primary.

The passive standby is generally not referred to as a cluster. The term *cluster* is reserved for multiple interconnected computers that are all actively doing processing while maintaining the image of a single system to the outside world. The term **active secondary** is often used in referring to this configuration. Three classifications of clustering can be identified: separate servers, shared nothing, and shared memory.

In one approach to clustering, each computer is a **separate server** with its own disks and there are no disks shared between systems (Figure 16.13a). This arrangement provides high performance as well as high availability. In this case, some type of management or scheduling software is needed to assign incoming client requests to servers so that the load is balanced and high utilization is achieved. It is desirable to have a failover capability, which means that if a computer fails while executing an application, another computer in the cluster can pick up and complete the application. For this to happen, data must constantly be copied among systems so that each system has access to the current data of the other systems. The overhead of this data exchange ensures high availability at the cost of a performance penalty.

To reduce the communications overhead, most clusters now consist of servers connected to common disks (Figure 16.13b). In one variation of this approach, called **shared nothing**, the common disks are partitioned into volumes, and each volume is owned by a single computer. If that computer fails, the cluster must be reconfigured so that some other computer has ownership of the volumes of the failed computer.

It is also possible to have multiple computers share the same disks at the same time (called the **shared disk** approach), so that each computer has access to all of the volumes on all of the disks. This approach requires the use of some type of locking facility to ensure that data can only be accessed by one computer at a time.

Operating System Design Issues

Full exploitation of a cluster hardware configuration requires some enhancements to a single-system operating system.

Failure Management How failures are managed by a cluster depends on the clustering method used (Table 16.2). In general, two approaches can be taken to dealing with failures: highly available clusters and fault-tolerant clusters. A highly available cluster offers a high probability that all resources will be in service. If a failure occurs, such as a node goes down or a disk volume is lost, then the queries in progress are lost. Any lost query, if retried, will be serviced by a different computer in the cluster. However, the cluster operating system makes no guarantee about the state of partially executed transactions. This would need to be handled at the application level.

A fault-tolerant cluster ensures that all resources are always available. This is achieved by the use of redundant shared disks and mechanisms for backing out uncommitted transactions and committing completed transactions.

The function of switching an application and data resources over from a failed system to an alternative system in the cluster is referred to as **failover**. A related function is the restoration of applications and data resources to the original system once it has been fixed; this is referred to as **failback**. Failback can be automated, but this is desirable only if the problem is truly fixed and unlikely to recur. If not, automatic failback can cause subsequently failed resources to bounce back and forth between computers, resulting in performance and recovery problems.

Load Balancing A cluster requires an effective capability for balancing the load among available computers. This includes the requirement that the cluster be incrementally scalable. When a new computer is added to the cluster, the load-balancing facility should automatically include this computer in scheduling applications. Middleware mechanisms need to recognize that services can appear on different members of the cluster and may migrate from one member to another.

Parallelizing Computation In some cases, effective use of a cluster requires executing software from a single application in parallel. [KAPP00] lists three general approaches to the problem:

- **Parallelizing compiler:** A parallelizing compiler determines, at compile time, which parts of an application can be executed in parallel. These are then split off to be assigned to different computers in the cluster. Performance depends on the nature of the problem and how well the compiler is designed.
- **Parallelized application:** In this approach, the programmer writes the application from the outset to run on a cluster and uses message passing to move data, as required, between cluster nodes. This places a high burden on the programmer but may be the best approach for exploiting clusters for some applications.
- **Parametric computing:** This approach can be used if the essence of the application is an algorithm or program that must be executed a large number of

times, each time with a different set of starting conditions or parameters. A good example is a simulation model, which will run a large number of different scenarios and then develop statistical summaries of the results. For this approach to be effective, parametric processing tools are needed to organize, run, and manage the jobs in an orderly manner.

Cluster Computer Architecture

Figure 16.14 shows a typical cluster architecture. The individual computers are connected by some high-speed LAN or switch hardware. Each computer is capable of operating independently. In addition, a middleware layer of software is installed in each computer to enable cluster operation. The cluster middleware provides a unified system image to the user, known as a **single-system image**. The middleware may also be responsible for providing high availability, by means of load balancing and responding to failures in individual components. [HWAN99] lists the following as desirable cluster middleware services and functions:

- **Single entry point:** A user logs onto the cluster rather than to an individual computer.
- **Single file hierarchy:** The user sees a single hierarchy of file directories under the same root directory.
- **Single control point:** There is a default node used for cluster management and control.
- **Single virtual networking:** Any node can access any other point in the cluster, even though the actual cluster configuration may consist of multiple interconnected networks. There is a single virtual network operation.
- **Single memory space:** Distributed shared memory enables programs to share variables.

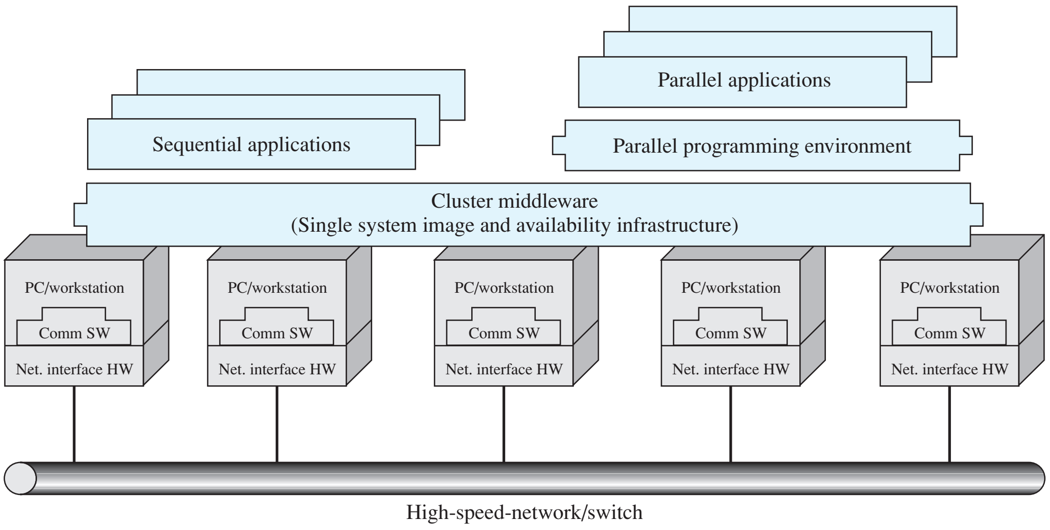

Parallel applications
Sequential applications
Parallel programming environment
Cluster middleware
(Single system image and availability infrastructure)
PC/workstation
Comm SW
Net. interface HW
High-speed-network/switch

Figure 16.14 Cluster Computer Architecture

- **Single job-management system:** Under a cluster job scheduler, a user can submit a job without specifying the host computer to execute the job.
- **Single user interface:** A common graphic interface supports all users, regardless of the workstation from which they enter the cluster.
- **Single I/O space:** Any node can remotely access any I/O peripheral or disk device without knowledge of its physical location.
- **Single process space:** A uniform process-identification scheme is used. A process on any node can create or communicate with any other process on a remote node.
- **Checkpointing:** This function periodically saves the process state and intermediate computing results, to allow rollback recovery after a failure.
- **Process migration:** This function enables load balancing.

The last four items on the preceding list enhance the availability of the cluster. The remaining items are concerned with providing a single system image.

Returning to Figure 16.14, a cluster will also include software tools for enabling the efficient execution of programs that are capable of parallel execution.

Clusters Compared to SMP

Both clusters and symmetric multiprocessors provide a configuration with multiple processors to support high-demand applications. Both solutions are commercially available, although SMP has been around far longer.

The main strength of the SMP approach is that an SMP is easier to manage and configure than a cluster. The SMP is much closer to the original single-processor model for which nearly all applications are written. The principal change required in going from a uniprocessor to an SMP is to the scheduler function. Another benefit of the SMP is that it usually takes up less physical space and draws less power than a comparable cluster. A final important benefit is that the SMP products are well established and stable.

Over the long run, however, the advantages of the cluster approach are likely to result in clusters dominating the high-performance server market. Clusters are far superior to SMPs in terms of incremental and absolute scalability. Clusters are also superior in terms of availability, because all components of the system can readily be made highly redundant.

16.5 WINDOWS CLUSTER SERVER

Windows Cluster Server (formerly code named Wolfpack) is a shared-nothing cluster, in which each disk volume and other resources are owned by a single system at a time.

The Windows Cluster Server design makes use of the following concepts:

- **Cluster Service:** The collection of software on each node that manages all cluster-specific activity.
- **Resource:** An item managed by the cluster service. All resources are objects representing actual resources in the system, including hardware devices such

as disk drives and network cards and logical items such as logical disk volumes, TCP/IP addresses, entire applications, and databases.

- **Online:** A resource is said to be online at a node when it is providing service on that specific node.
- **Group:** A collection of resources managed as a single unit. Usually, a group contains all of the elements needed to run a specific application and for client systems to connect to the service provided by that application.

The concept of group is of particular importance. A group combines resources into larger units that are easily managed, both for failover and load balancing. Operations performed on a group, such as transferring the group to another node, automatically affect all of the resources in that group. Resources are implemented as dynamically linked libraries (DLLs) and managed by a resource monitor. The resource monitor interacts with the cluster service via remote procedure calls and responds to cluster service commands to configure and move resource groups.

Figure 16.15 depicts the Windows Cluster Server components and their relationships in a single system of a cluster. The **node manager** is responsible for maintaining this node's membership in the cluster. Periodically, it sends heartbeat messages to the node managers on other nodes in the cluster. In the event that one node manager detects a loss of heartbeat messages from another cluster node, it

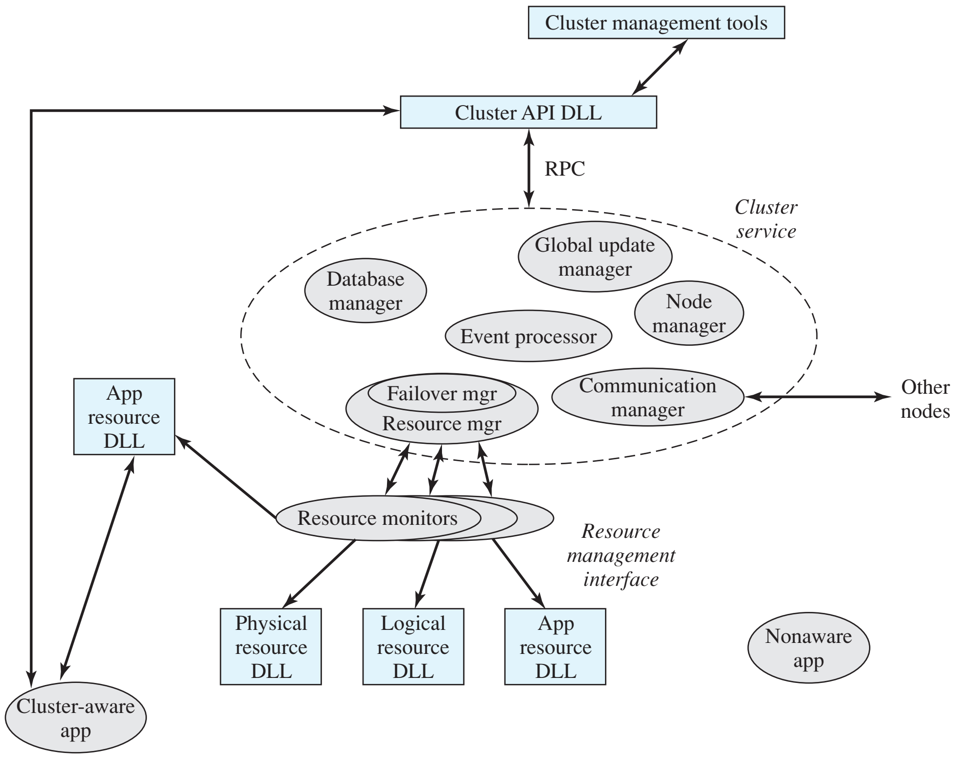


Figure 16.15 Windows Cluster Server Block Diagram

broadcasts a message to the entire cluster, causing all members to exchange messages to verify their view of current cluster membership. If a node manager does not respond, it is removed from the cluster and its active groups are transferred to one or more other active nodes in the cluster.

The **configuration database manager** maintains the cluster configuration database. The database contains information about resources and groups and node ownership of groups. The database managers on each of the cluster nodes cooperate to maintain a consistent picture of configuration information. Fault-tolerant transaction software is used to assure that changes in the overall cluster configuration are performed consistently and correctly.

The **resource manager/failover manager** makes all decisions regarding resource groups and initiates appropriate actions such as startup, reset, and failover. When failover is required, the failover managers on the active node cooperate to negotiate a distribution of resource groups from the failed system to the remaining active systems. When a system restarts after a failure, the failover manager can decide to move some groups back to this system. In particular, any group may be configured with a preferred owner. If that owner fails and then restarts, the group is moved back to the node in a rollback operation.

The **event processor** connects all of the components of the cluster service, handles common operations, and controls cluster service initialization. The communications manager manages message exchange with all other nodes of the cluster. The global update manager provides a service used by other components within the cluster service.

16.6 SUN CLUSTER

Sun Cluster is a distributed operating system built as a set of extensions to the base Solaris UNIX system. It provides cluster with a single-system image; that is, the cluster appears to the user and applications as a single computer running the Solaris operating system.

Figure 16.16 shows the overall architecture of Sun Cluster. The major components are

- Object and communication support
- Process management
- Networking
- Global distributed file system

Object and Communication Support

The Sun Cluster implementation is object oriented. The CORBA object model (see Appendix B) is used to define objects and the remote procedure call (RPC) mechanism implemented in Sun Cluster. The CORBA Interface Definition Language (IDL) is used to specify interfaces between MC components in different nodes. The elements of MC are implemented in the object-oriented language C++. The use of a uniform object model and IDL provides a mechanism for internode and intranode

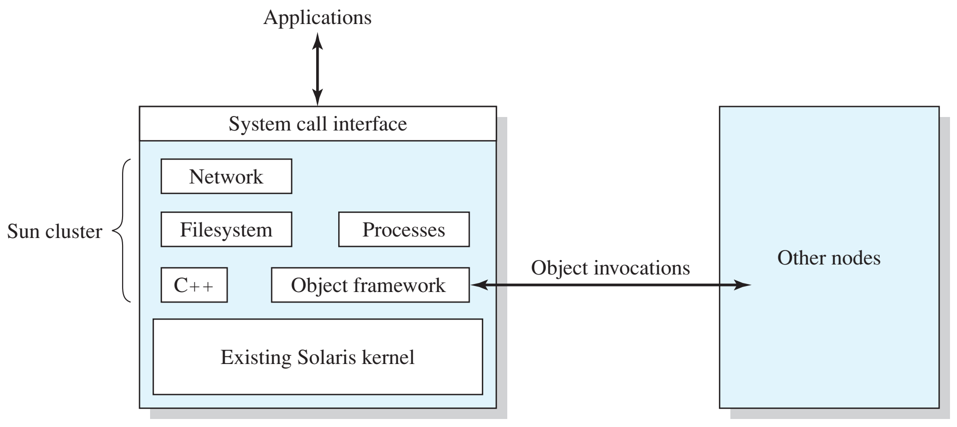


Figure 16.16 **Sun Cluster Structure**

interprocess communication. All of this is built on top of the Solaris kernel with virtually no changes required to the kernel.

Process Management

Global process management extends process operations so that the location of a process is transparent to the user. Sun Cluster maintains a global view of processes so that there is a unique identifier for each process in the cluster and so that each node can learn the location and status of each process. Process migration (described in Chapter 18) is possible: A process can move from one node to another during its lifetime, to achieve load balancing or for failover. However, the threads of a single process must be on the same node.

Networking

The designers of Sun Cluster considered three approaches for handling network traffic:

1. Perform all network protocol processing on a single node. In particular, for a TCP/IP-based application, incoming (and outgoing) traffic would go through a network-connection node that for incoming traffic would analyze TCP and IP headers and route the encapsulated data to the appropriate node; and for outgoing traffic would encapsulate data from other nodes in TCP/IP headers. This approach is not scalable to a large number of nodes and so was rejected.
2. Assign a unique IP address to each node and run the network protocols over the external network directly to each node. One difficulty with this approach is that the cluster configuration is no longer transparent to the outside world. Another complication is the difficulty of failover when a running application moves to another node with a different underlying network address.
3. Use a packet filter to route packets to the proper node and perform protocol processing on that node. Externally, the cluster appears as a single server with a single IP address. Incoming connections (client requests) are load balanced among the available nodes of the cluster. This is the approach adopted in Sun Cluster.

The Sun Cluster networking subsystem has three key elements:

1. Incoming packets are first received on the node that has the network adapter physically attached to it; the receiving node filters the packet and delivers it to the correct target node over the cluster interconnect.
2. All outgoing packets are routed over the cluster interconnect to the node (or one of multiple alternative nodes) that has an external network physical connection. All protocol processing for outgoing packets is done by the originating node.
3. A global network configuration database is maintained to keep track of network traffic to each node.

Global File System

The most important element of Sun Cluster is the global file system, depicted in Figure 16.17, which contrasts MC file management with the basic Solaris scheme. Both are built on the use of vnode and virtual file system concepts.

In Solaris, the virtual node (vnode) structure is used to provide a powerful, general-purpose interface to all types of file systems. A vnode is used to map pages of memory into the address space of a process and to permit access to a file system. While an inode is used to map processes to UNIX files, a vnode can map a process to an object in any file system type. In this way, a system call need not understand the actual object being manipulated, only how to make the proper object-oriented type call using the vnode interface. The vnode interface accepts general-purpose file manipulation commands, such as read and write, and translates them into actions appropriate for the subject file system. Just as vnodes are used to describe individual file system objects, the virtual file system (vfs) structures are used to describe entire file systems. The vfs interface accepts general-purpose commands that operate on entire files and translates them into actions appropriate for the subject file system.

In Sun Cluster, the global file system provides a uniform interface to files distributed over the cluster. A process can open a file located anywhere in the cluster,

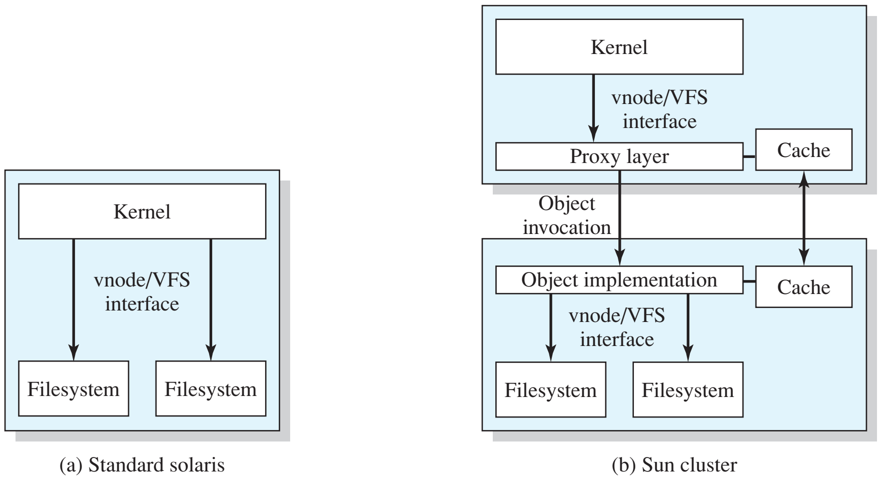


Figure 16.17 Sun Cluster File System Extensions

and processes on all nodes use the same pathname to locate a file. To implement global file access, MC includes a proxy file system built on top of the existing Solaris file system at the vnode interface. The vfs/vnode operations are converted by a proxy layer into object invocations (see Figure 16.17b). The invoked object may reside on any node in the system. The invoked object performs a local vnode/vfs operation on the underlying file system. Neither the kernel nor the existing file systems have to be modified to support this global file environment.

To reduce the number of remote object invocations, caching is used. Sun Cluster supports caching of file contents, directory information, and file attributes.

16.7 BEOWULF AND LINUX CLUSTERS

In 1994, the Beowulf project was initiated under the sponsorship of the NASA High Performance Computing and Communications (HPCC) project. Its goal was to investigate the potential of clustered PCs for performing important computation tasks beyond the capabilities of contemporary workstations at minimum cost. Today, the Beowulf approach is widely implemented and is perhaps the most important cluster technology available.

Beowulf Features

Key features of Beowulf include the following [RIDG97]:

- Mass market commodity components
- Dedicated processors (rather than scavenging cycles from idle workstations)
- A dedicated, private network (LAN or WAN or internetted combination)
- No custom components
- Easy replication from multiple vendors
- Scalable I/O
- A freely available software base
- Use of freely available distribution computing tools with minimal changes
- Return of the design and improvements to the community

Although elements of Beowulf software have been implemented on a number of different platforms, the most obvious choice for a base is Linux, and most Beowulf implementations use a cluster of Linux workstations and/or PCs. Figure 16.18 depicts a representative configuration. The cluster consists of a number of workstations, perhaps of differing hardware platforms, all running the Linux operating system. Secondary storage at each workstation may be made available for distributed access (for distributed file sharing, distributed virtual memory, or other uses). The cluster nodes (the Linux systems) are interconnected with a commodity networking approach, typically Ethernet. The Ethernet support may be in the form of a single Ethernet switch or an interconnected set of switches. Commodity Ethernet products at the standard data rates (10 Mbps, 100 Mbps, 1 Gbps) are used.

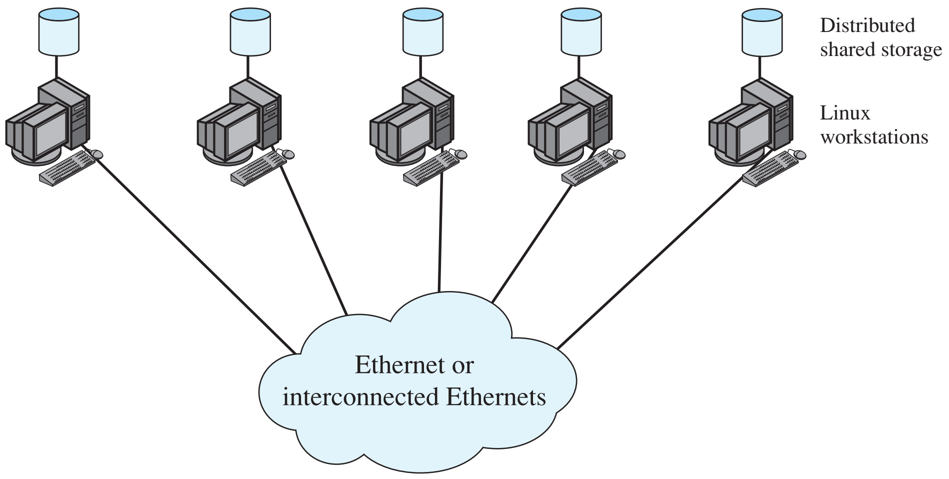

Distributed shared storage
Linux workstations
Ethernet or interconnected Ethernets

Figure 16.18 **Generic Beowulf Configuration**

Beowulf Software

The Beowulf software environment is implemented as an add-on to commercially available, royalty-free base Linux distributions. The principal source of open-source Beowulf software is the Beowulf site at www.beowulf.org, but numerous other organizations also offer free Beowulf tools and utilities.

Each node in the Beowulf cluster runs its own copy of the Linux kernel and can function as an autonomous Linux system. To support the Beowulf cluster concept, extensions are made to the Linux kernel to allow the individual nodes to participate in a number of global namespaces. The following are examples of Beowulf system software:

- **Beowulf distributed process space (BPROC):** This package allows a process ID space to span multiple nodes in a cluster environment and also provides mechanisms for starting processes on other nodes. The goal of this package is to provide key elements needed for a single system image on Beowulf cluster. BPROC provides a mechanism to start processes on remote nodes without ever logging into another node and by making all the remote processes visible in the process table of the cluster's front-end node.
- **Beowulf Ethernet Channel Bonding:** This is a mechanism that joins multiple low-cost networks into a single logical network with higher bandwidth. The only additional work over using single network interface is the computationally simple task of distributing the packets over the available device transmit queues. This approach allows load balancing over multiple Ethernets connected to Linux workstations.
- **Pvmsync:** This is a programming environment that provides synchronization mechanisms and shared data objects for processes in a Beowulf cluster.

- **EnFuzion:** EnFuzion consists of a set of tools for doing parametric computing, as described in Section 16.4. Parametric computing involves the execution of a program as a large number of jobs, each with different parameters or starting conditions. EnFusion emulates a set of robot users on a single root node machine, each of which will log into one of the many clients that form a cluster. Each job is set up to run with a unique, programmed scenario, with an appropriate set of starting conditions [KAPP00].

16.8 SUMMARY

Client/server computing is the key to realizing the potential of information systems and networks to improve productivity significantly in organizations. With client/server computing, applications are distributed to users on single-user workstations and personal computers. At the same time resources that can and should be shared are maintained on server systems that are available to all clients. Thus, the client/server architecture is a blend of decentralized and centralized computing.

Typically, the client system provides a graphical user interface (GUI) that enables a user to exploit a variety of applications with minimal training and relative ease. Servers support shared utilities, such as database management systems. The actual application is divided between client and server in a way intended to optimize ease of use and performance.

The key mechanism required in any distributed system is interprocess communication. Two techniques are in common use. A message-passing facility generalizes the use of messages within a single system. The same sorts of conventions and synchronization rules apply. Another approach is the use of the remote procedure call. This is a technique by which two programs on different machines interact using procedure call/return syntax and semantics. Both the called and calling program behave as if the partner program were running on the same machine.

A cluster is a group of interconnected, whole computers working together as a unified computing resource that can create the illusion of being one machine. The term *whole computer* means a system that can run on its own, apart from the cluster.

16.9 RECOMMENDED READING AND WEB SITES

[SING99] provides good coverage of the topics in this chapter. [BERS96] provides a good technical discussion of the design issues involved in allocating applications to client and server and in middleware approaches; the book also discusses products and standardization efforts. A good overview of middleware technology and products is [BRIT04]. [MENA05] provides a performance comparison of remote procedure calls and distributed message passing.

[TANE85] is a survey of distributed operating systems that covers both distributed process communication and distributed process management. [CHAN90] provides an overview of distributed message passing operating systems. [TAY90] is a survey of the approach taken by various operating systems in implementing remote procedure calls.

A thorough treatment of clusters can be found in [BUYY99a] and [BUYY99b]. The former has a good treatment of Beowulf, which is also nicely covered in

[RIDG97]. A more detailed treatment of Beowulf is [STER99].Windows Cluster Server is described in [SHOR97]; [RAJA00] provides a more detailed treatment. Sun Cluster is described in [SUN99] and [KHAL96]. [LAI06] provides a close examination of thin client architecture.

BERS96 Berson, A. *Client/Server Architecture.* New York: McGraw-Hill, 1996.

BRIT04 Britton, C. *IT Architectures and Middleware.* Reading, MA: Addison-Wesley, 2004.

BUYY99a Buyya, R. *High Performance Cluster Computing: Architectures and Systems.* Upper Saddle River, NJ: Prentice Hall, 1999.

BUYY99b Buyya, R. *High Performance Cluster Computing: Programming and Applications.* Upper Saddle River, NJ: Prentice Hall, 1999.

CHAN90 Chandras, R. "Distributed Message Passing Operating Systems." *Operating Systems Review*, January 1990.

KHAL96 Khalidi, Y., et al. "Solaris MC: A Multicomputer OS." Proceedings, 1996 USENIX Conference, January 1996.

LAI06 Lai, A., and Nieh, J. "On the Performance of Wide-Area Thin-Client Computing." *ACM Transactions on Computer Systems*, May 2006.

MENA05 Menasce, D. "MOM vs. RPC: Communication Models for Distributed Applications." *IEEE Internet Computing*, March/April 2005.

RAJA00 Rajagopal, R. *Introduction to Microsoft Windows NT Cluster Server.* Boca Raton, FL: CRC Press, 2000.

RIDG97 Ridge, D., et al. "Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs." *Proceedings, IEEE Aerospace*, 1997.

SHOR97 Short, R.; Gamache, R.; Vert, J.; and Massa, M. "Windows NT Clusters for Availability and Scalability." *Proceedings, COMPCON Spring 97*, February 1997.

SING99 Singh, H. *Progressing to Distributed Multiprocessing.* Upper Saddle River, NJ: Prentice Hall, 1999.

STER99 Sterling, T., et al. *How to Build a Beowulf.* Cambridge, MA: MIT Press, 1999.

SUN99 Sun Microsystems. "Sun Cluster Architecture: A White Paper." *Proceedings, IEEE Computer Society International Workshop on Cluster Computing*, December 1999.

TANE85 Tanenbaum, A., and Renesse, R. "Distributed Operating Systems." *Computing Surveys*, December 1985.

TAY90 Tay, B., and Ananda, A. "A Survey of Remote Procedure Calls." *Operating Systems Review*, July 1990.

Recommended Web sites:

- **SQL Standards:** A central source of information about the SQL standards process and its current documents
- **IEEE Computer Society Task Force on Cluster Computing:** An international forum to promote cluster computing research and education
- **Beowulf:** An international forum to promote cluster computing research and education

16.10 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

applications programming interface (API)	distributed message passing	message
Beowulf	failback	middleware
client	failover	remote procedure call (RPC)
client/server	fat client	server thin client
cluster	file cache consistency graphical user interface (GUI)	

Review Questions

16.1 What is client/server computing?

16.2 What distinguishes client/server computing from any other form of distributed data processing?

16.3 What is the role of a communications architecture such as TCP/IP in a client/server environment?

16.4 Discuss the rationale for locating applications on the client, the server, or split between client and server.

16.5 What are fat clients and thin clients, and what are the differences in philosophy of the two approaches?

16.6 Suggest pros and cons for fat client and thin client strategies.

16.7 Explain the rationale behind the three-tier client/server architecture.

16.8 What is middleware?

16.9 Because we have standards such as TCP/IP, why is middleware needed?

16.10 List some benefits and disadvantages of blocking and nonblocking primitives for message passing.

16.11 List some benefits and disadvantages of nonpersistent and persistent binding for RPCs.

16.12 List some benefits and disadvantages of synchronous and asynchronous RPCs.

16.13 List and briefly define four different clustering methods.

Problems

16.1 Let α be the percentage of program code that can be executed simultaneously by n computers in a cluster, each computer using a different set of parameters or initial conditions. Assume that the remaining code must be executed sequentially by a single processor. Each processor has an execution rate of x MIPS.

a. Derive an expression for the effective MIPS rate when using the system for exclusive execution of this program, in terms of n, α, and x.

b. If $n = 16$ and $x = 4$ MIPS, determine the value of α that will yield a system performance of 40 MIPS.

16.2 An application program is executed on a 9-computer cluster. A benchmark program takes time T on this cluster. Further, 25% of T is time in which the application is running simultaneously on all 9 computers. The remaining time, the application has to run on a single computer.

a. Calculate the effective speedup under the aforementioned condition as compared to executing the program on a single computer. Also calculate α, the percentage of code that has been parallelized (programmed or compiled so as to use the cluster mode) in the preceding program.

b. Suppose that we are able to effectively use 18 computers rather than 9 computers on the parallelized portion of the code. Calculate the effective speedup that is achieved.

16.3 The following FORTRAN program is to be executed on a computer, and a parallel version is to be executed on a 32-computer cluster.

```
L1:       DO 10 I = 1, 1024
L2:       SUM(I) = 0
L3:       DO 20 J = 1, I
L4:   20  SUM(I) = SUM(I) + I
L5:   10   CONTINUE
```

Suppose lines 2 and 4 each take two machine cycle times, including all processor and memory-access activities. Ignore the overhead caused by the software loop control statements (lines 1, 3, 5) and all other system overhead and resource conflicts.

a. What is the total execution time (in machine cycle times) of the program on a single computer?

b. Divide the I-loop iterations among the 32 computers as follows: Computer 1 executes the first 32 iterations (I = 1 to 32), processor 2 executes the next 32 iterations, and so on. What are the execution time and speedup factor compared with part (a)? (Note that the computational workload, dictated by the J-loop, is unbalanced among the computers.)

c. Explain how to modify the parallelizing to facilitate a balanced parallel execution of all the computational workload over 32 computers. *Balanced load* means an equal number of additions assigned to each computer with respect to both loops.

d. What is the minimum execution time resulting from the parallel execution on 32 computers? What is the resulting speedup over a single computer?