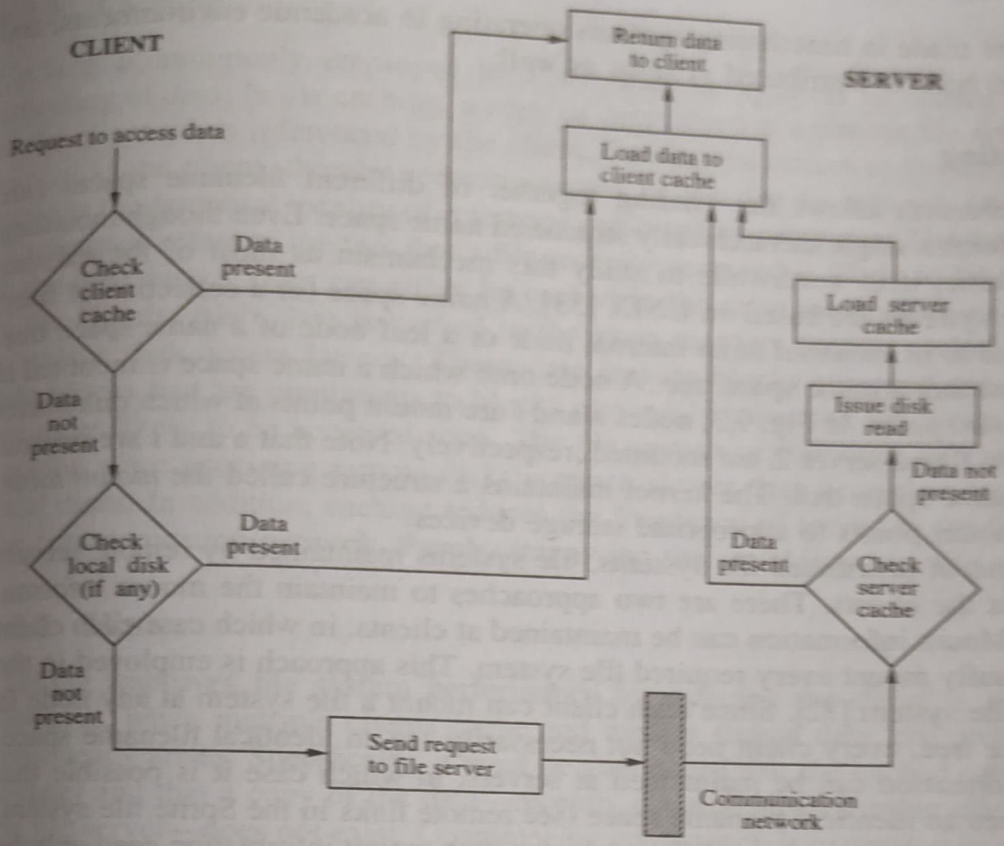

FIGURE 9.2
Typical data access actions in distributed file systems.

and retrieval operations. Typically, data access in a distributed file system proceeds as shown in Fig. 9.2.

A request by a process to access a data block is presented to the local cache (client cache) of the machine (client) on which the process is running (see Fig. 9.1). If the block is not in the cache, then the local disk, if present, is checked for the presence of the data block. If the block is present, then the request is satisfied and the block is loaded into the client cache. If the block is not stored locally, then the request is passed on to the appropriate file server (as determined by the name server). The server checks its own cache for the presence of the data block before issuing a disk I/O request. The data block is transferred to the client cache in any case and loaded to the server cache if it was missing in the server cache.

9.3 MECHANISMS FOR BUILDING DISTRIBUTED FILE SYSTEMS

In this section, the basic mechanisms underlying the majority of the distributed file systems operating today are presented [33]. These mechanisms take advantage of the observations made in previous studies on file systems. We cite these observations along with the mechanisms that exploit them. A crucial point to note here is that these ob-

overhead and disk seek time over many consecutive blocks of a file. Bulk transfers reduce file access overhead through obtaining a multiple number of blocks with a single seek; by formatting and transmitting a multiple number of large packets in a single context switch; and by reducing the number of acknowledgments that need to be sent. Bulk transfers exploit the fact that most files are accessed in their entirety [33].

9.3.5 Encryption

Encryption is used for enforcing security in distributed systems [33]. The work of Needham and Schroeder [23] is the basis for most of the current security mechanisms in distributed systems (see Sec. 15.8). In their scheme, two entities wishing to communicate with each other establish a key for conversation with the help of an authentication server. It is important to note that the conversation key is determined by the authentication server, but is never sent in plain (unencrypted) text to either of the entities.

9.4 DESIGN ISSUES

We now discuss various issues that must be addressed in the design and implementation of distributed file systems. By studying these design issues, one can better understand the intricacies of a distributed file system.

9.4.1 Naming and Name Resolution

A name in file systems is associated with an object (such as a file or a directory). *Name resolution* refers to the process of mapping a name to an object or, in the case of replication, to multiple objects. A *name space* is a collection of names which may or may not share an identical resolution mechanism.

Traditionally, there have been three approaches to name files in a distributed environment [29]. The simplest scheme is to concatenate the host name to the names of files that are stored on that host. While this approach guarantees that a filename is unique systemwide, it conflicts with the goal of network transparency. Another serious problem with this approach is that moving a file from one host to another requires changes in the filename and in the applications accessing that file. That is, this naming scheme is not *location-independent*. (If a naming scheme is location-independent, the name of a file need not be changed when the file's physical storage location changes [18].) The main advantage of this scheme, however, is that name resolution is very simple as a file can be located without consulting any other host in the system.

The second approach is to mount remote directories onto local directories. (See Sec. 9.3.1 for details.) Mounting a remote directory requires that the host of the directory be known. Once a remote directory is mounted, its files can be referenced in a *location-transparent* manner. (A naming scheme is said to be location-transparent if the name of a file does not reveal any hint as to its physical storage location [18].) This approach can also resolve a filename without consulting any host.

The third approach is to have a single global directory where all the files in the system belong to a single name space. Variations of this scheme are found in the Sprite

and Apollo systems (see Sec. 9.5). This approach does not have the disadvantages of the above two naming schemes. The main disadvantage of this scheme, however, is that it is mostly limited to one computing facility or to a few cooperating computing facilities. This limitation is due to the requirement of systemwide unique filenames, which requires that all the computing facilities involved cooperate [6]. Thus, this scheme is impractical for distributed systems that encompass heterogeneous environments and wide geographical areas, where a naming scheme suitable for one computing facility may be unsuitable for another.

THE CONCEPT OF CONTEXTS. To overcome the difficulties associated with systemwide unique names, the notion of *context* has been used to partition a name space. A context identifies the name space in which to resolve a given name. Contexts can partition a name space along the following: geographical boundary, organizational boundary, specific to hosts, a file system type, etc. In a context based scheme, a filename can be thought of as composed of a context and a name local to that context. Resolving a name involves interpreting the name with respect to the given context. The interpretation may be complete within the given context or may lead to yet another context, in which case the above process is repeated. If all files share a common initial context, then unique systemwide global names result.

The x-Kernel logical file system (see Sec. 9.5.5) is a file system that makes use of contexts. In this file system, a user defines his own file space hierarchy. The internal nodes in this hierarchy correspond to the contexts.

The *Tilde* naming scheme is another variant of the naming scheme using contexts [6]. In the Tilde naming scheme, the name space is partitioned (based on projects which people are associated with) into a set of logically independent directory trees called *tilde trees*. Each process running in the system has a set of tilde trees associated with it that constitute the process's tilde environment. When a process tries to open or manipulate a file, the filename is interpreted with respect to the process's tilde environment.

NAME SERVER. In a centralized system, name resolution can be accomplished by maintaining a table that maps names to objects. In distributed systems, *name servers* are responsible for name resolution. A name server is a process that maps names specified by clients to stored objects such as files and directories. The easiest approach to name resolution in distributed systems is for all clients to send their queries to a single name server which maps names to objects. This approach has the following serious drawbacks: first, if the name server crashes, the entire system is drastically affected. Second, the name server may become a bottleneck and seriously degrade the performance of the system.

The second approach involves having several name servers (on different hosts) wherein each server is responsible for mapping objects stored in different domains. This approach is commonly used in the distributed file systems operating today. When a name (usually with many components such as "a/b/c") is to be mapped to an object, the local name server (such as a table maintained in the kernel) is queried. The local name server may point to a remote name server for further mapping of the name. For example, querying /a/b/c may require a remote server mapping the /b/c part of

the filename. This procedure is repeated until the name is completely resolved. By replicating the tables used by name servers, one can achieve fault tolerance and higher performance.

9.4.2 Caches on Disk or Main Memory

The benefits obtained by employing file caches at clients were discussed in Sec. 9.3.2. This section is concerned with the question of whether the data cached by a client should be in the main memory at the client or on a local disk at the client. The advantages of having the cache in the main memory are as follows [24]:

- Diskless workstations can also take advantage of caching. (Note that diskless workstations are cheaper.)
- Accessing a cache in main memory is much faster than accessing a cache on local disk.
- The server-cache is in the main memory at the server, and hence a single design for a caching mechanism is applicable to both clients and servers.

The main disadvantage of having client-cache in main memory is that it competes with the virtual memory system for physical memory space. Thus, a scheme to deal with the memory contention between cache and virtual memory system is necessary. This scheme should also prevent data blocks from being present in both the virtual memory and the cache. A consequence of this fact is a more complex cache manager and memory management system. A limitation of caching in main memory is that large files cannot be cached completely in main memory, thus requiring the caching to be block-oriented. Block-oriented caching is more complex and imposes more load at the file servers (see Bulk Data Transfer) relative to entire file caching.

The advantages of caching on a local disk are: large files can be cached without affecting a workstation's performance; the virtual memory management is simple; and it facilitates the incorporation of portable workstations into a distributed system (see Coda, Section 9.5.4). A workstation, before being disconnected from the network for portable use, will cache all the required files onto its local disk.

9.4.3 Writing Policy

The writing policy decides when a modified cache block at a client should be transferred to the server. The simplest policy is *write-through*. In write-through, all writes requested by the applications at clients are also carried out at the servers immediately. The main advantage of write-through is reliability. In the event of a client crash, little information is lost. A write-through policy, however, does not take advantage of the cache.

An alternate writing policy, *delayed writing policy*, delays the writing at the server [24]. In this case, modifications due to a write are reflected at the server after some delay. This approach can potentially take advantage of the cache by performing many writes on a block present locally in the cache. Another motivation for delaying the writes is that some of the data (for example, intermediate results) could be deleted

in a short time, in which case data need not be written at the server at all. In fact, it has been reported that twenty to thirty percent of new data is deleted within thirty seconds [26]. One factor that needs to be taken into account when deciding the length of the delay period is the likelihood of a block not being modified after a given period. While the delayed writing policy takes advantage of the cache at a client, it introduces the reliability problem. In the event of a client crash, a significant amount of data can be lost.

Another writing policy delays the updating of the files at the server until the file is closed at the client. In this policy, the traffic at the server depends on the average period that files are open. If the average period for which files are open is short, then this policy does not greatly benefit from delaying the updates. On the other hand, if the average period for which files are open is long, this policy is also susceptible to losing data in the event of a client crash. Note that it has been reported that a majority of the files are open for a very short time [26].

9.4.4 Cache Consistency

The problem of cache consistency was introduced in Sec. 9.3.3. This section is concerned with the schemes that can guarantee consistency of the data cached at clients. There are two approaches to guarantee that the data returned to the clients is valid [42].

- In the *server-initiated* approach, servers inform cache managers whenever the data in the client caches become stale. Cache managers at clients can then retrieve the new data or invalidate the blocks containing the old data in their cache.
- In the *client-initiated* approach, it is the responsibility of the cache managers at the clients to validate data with the server before returning it to the clients.

Both of these approaches are expensive and unattractive as they require elaborate cooperation between servers and cache managers. In both approaches, communication costs are high. The server-initiated approach requires the server to maintain reliable records on what data blocks are cached by which cache managers. The client-initiated approach simply negates the benefit of having a cache by checking the server to validate data on every access. It also does not scale well, as the load at the server caused by client checking increases with the increase in the number of clients.

A third approach for cache consistency is simply not to allow file caching when *concurrent-write sharing* occurs. In concurrent-write sharing, a file is open at multiple clients and at least one client has it open for writing [24]. In this approach, the file server has to keep track of the clients sharing a file. When concurrent-write sharing occurs for a file, the file server informs all the clients to purge their cached data items belonging to that file. Alternatively, concurrent-write sharing can be avoided by locking files (see the Apollo file system, Sec. 9.5.3).

Another issue that a cache consistency scheme needs to address is *sequential-write sharing*, which occurs when a client opens a file that has recently been modified and closed by another client [24]. Two potential problems with sequential-write sharing are: (1) when a client opens a file, it may have outdated blocks of the file in its cache, and

(2) when a client opens a file, the current data blocks may still be in another client's cache waiting to be flushed. This can happen when the delayed writing policy is used.

To handle the first problem, files usually have timestamps associated with them. When data blocks of a file are cached, the timestamp associated with the file is also cached. An inconsistency can be detected by comparing the timestamp of the cached data block with the timestamp of the file at the server.

To handle the second problem, the server must require that clients flush the modified blocks of a file from their cache whenever a new client opens the file for writing.

9.4.5 Availability

Availability is one of the important issues in the design of distributed file systems. The failure of servers or the communication network can severely affect the availability of files. *Replication* is the primary mechanism used for enhancing the availability of files in distributed file systems.

REPLICATION. Under replication, many copies or replicas of files are maintained at different servers. Replication is inherently expensive because of the extra storage space required to store the replicas and the overhead incurred in maintaining all the replicas up to date. The most serious problems with replication are (1) how to keep the replicas of a file consistent and (2) how to detect inconsistencies among replicas of a file and subsequently recover from these inconsistencies. Some typical situations that cause inconsistency among replicas are (a) a replica is not updated due to the failure of the server storing the replica and (b) all the file servers storing the replicas of a file are not reachable from all the clients due to network partition, and the replicas of a file in different partitions are updated differently. *Ironically, potential inconsistency problems may preclude file updates, thereby decreasing the availability as the level of replication is increased.*

UNIT OF REPLICATION. A fundamental design issue in replication is the *unit* of replication. The most basic unit is a *file*. File is the most commonly used replication unit and has been used in the Roe [9], Sprite [25], and Cedar [40] file systems. While this unit allows the replication of only those files that need to have higher availability, it makes overall replica management harder. For example, the protection rights associated with a directory have to be individually stored with each replica; replicas of files belonging to a common directory may not have common file servers and hence require extra name resolutions to locate the replicas in the case of modifications to the directory or the file.

Alternatively, the replication unit can be a group of all the files of a single user or the files that are in a server, etc. The group of files is referred to as a *volume* [38]. This scheme is used in Coda (see Sec. 9.5.4). The main advantage of volume replication is that replica management is easier. Protection rights can be associated with the volume instead of with each individual file replica. However, volume replication may be wasteful as a user typically needs higher availability for only a few files in the volume.

A compromise between volume replication and single file replication, used in Locus [43], captures the advantages of the above two schemes. In this scheme, all the

files of a user constitute a filegroup called a *primary pack*. A replica of a primary pack, called a pack, is allowed to contain a subset of the files in the primary pack. With this arrangement, a different degree of replication for each file in the primary pack can be obtained by creating one or more packs of the primary pack.

REPLICA MANAGEMENT. Replica management is concerned with the maintenance of replicas and in making use of them to provide increased availability. Replica management depends on whether consistency is guaranteed by the distributed file system. Here we are concerned with the consistency among replicas only (which is also known as *mutual consistency*), and not with the consistency within a file. The consistency within a file was discussed in Secs. 9.3.3 and 9.4.4.

To ensure mutual consistency among replicas, a weighted voting scheme can be used. We explain this scheme only briefly here as it is discussed in detail in Sec. 13.6. In this scheme, some number of votes and a timestamp are associated with each replica. A certain number of votes r or w must be obtained before a read or write, respectively, can be performed. Only votes from current (i.e., up-to-date) copies are valid. Reads can be from any current copy and writes update all the current copies. Timestamps of all the participating replicas (i.e., only current copies) are updated when a copy is updated. By keeping $w >$ 'total number of votes' of all the replicas, it is possible to maintain at least one current copy. An important point to observe is that it is not necessary to keep all replicas up-to-date as long as sufficient votes can be obtained to perform reads and writes. This key feature provides for increased availability and fault tolerance during system failures. Voting is used in the Roe file system [9] to maintain mutual consistency.

Another scheme to maintain consistency among replicas is to designate one or more processes as agents for controlling the access to replicas of files. This approach has been used in Locus [43]. In Locus, each filegroup has a designated site that enforces the global synchronization policy. This designated site is referred to as the *current synchronization site* (CSS). The file open and file close requests are routed through the CSS to a storage site which has the copy of the requested file. A disadvantage of this approach is that the agent processes can potentially become bottlenecks; hence it has poor scalability.

In the Harp file system [19], the designated site (server) for controlling the access to replicas is referred to as primary, and the other sites (servers) are referred to as backups. The primary enforces the global synchronization policy to maintain consistency in consultation with the backups.

In the Coda file system, mutual consistency among replicas is not assured. For details on its replica management, see Sec. 9.5.4.

9.4.6 Scalability

The issue of scalability deals with the suitability of the design of a system to cater to the demands of a growing system. Currently, client-server organization is a commonly used approach to structure distributed file systems (see Case Studies, Sec. 9.5). Caching, which reduces network latency and server-load, is the primary technique used in client-

server organization to improve the client response time. Caching, however, introduces the cache consistency problem, as many clients can cache a file.

Server-initiated cache invalidation is the most commonly used approach to maintain cache consistency. In this scheme, a server keeps track of all the clients sharing files stored on the server. This information forms a part of the server state. As the system grows larger, both the size of the server state and the load due to invalidations increase.

The server state and the server load can be reduced by exploiting knowledge about the usage of files [4, 34]. An important observation in this regard is that many widely used and shared files are accessed in read-only mode. Note that there is no need to check the validity (stale or up-to-date) of these files or to maintain the list of clients at servers for invalidation purposes.

Another observation is that the data required by a client is often found in another client's cache [4]. Since clients have more free cycles compared to servers [34], a client can obtain required data from another client rather than from a server. This of course raises the question of how to find the client which has cached the required data.

Blaze and Alonso [3] have proposed a scheme wherein a server serves (providing data and invalidating in case of updates) only Δ number of clients for a file at any time. New clients after the first Δ clients are informed of the Δ clients from whom they can obtain data. These Δ clients will also serve Δ number of clients, after which the new clients are informed of the identity of the Δ clients they served, and so on. The hierarchy of who is serving who forms a tree of maximum degree Δ. Cache misses and invalidation messages propagate up-and-down in this hierarchy where each internal node serves as a mini-file server for its children.

The structure of the server process also plays a major role in deciding how many clients a server can support. If the server is designed with a single process, then many clients have to wait for a long time whenever a disk I/O is initiated. These waits can be avoided if a separate process is assigned to each client. In this case, however, significant overhead due to the frequent context switches to handle requests from different clients can slow down the server. Lightweight processes (threads) have been proposed to reduce the context switch overhead. Threads are discussed in greater detail in Sec. 17.4.

9.4.7 Semantics

The semantics of a file system characterizes the effects of accesses on files. The basic semantics easily understood and easy to handle by programmers is that a read operation will return the data (stored) due to the latest write operation.

Guaranteeing the above semantics in distributed file systems, which employ caching, is difficult and expensive. Consider a file system employing server-initiated cache invalidation for the guarantee of cache consistency. In such a system, because of communication delays, invalidations may not occur immediately after updates and before reads occur at clients. To guarantee the above semantics, all the reads and writes from various clients will have to go through the server, or sharing will have to be disallowed either by the server, or by the use of locks by applications. Observe that in the first approach, the server can potentially become a bottleneck and the overheads are high because of

high traffic between the server and the clients. In the latter approach, however, the file is not available for certain clients.

9.5 CASE STUDIES

In the following sections, we describe

CHAPTER 10

DISTRIBUTED SHARED MEMORY

10.1 INTRODUCTION

Traditionally, distributed computing has been based on the message passing model in which processes interact and share data with each other by exchanging data in the form of messages. Hoare's communicating sequential processes (Sec. 2.6.4), the client-server model (Sec. 4.5.9), and remote procedure calls (Sec. 4.7.2) are examples of this model.

Distributed shared memory (DSM) system is a resource management component of a distributed operating system that implements the shared memory model in distributed systems, which have no physically shared memory (Fig. 10.1). The shared memory model provides a virtual address space that is shared among all nodes (computers) in a distributed system.

10.2 ARCHITECTURE AND MOTIVATION

With DSM, programs access data in the shared address space just as they access data in traditional virtual memory. In systems that support DSM, data moves between secondary memory and main memory as well as between main memories of different nodes. Each node can own[†] data stored in the shared address space, and the ownership can change

[†]Typically, the node which creates a data object owns the data object initially.

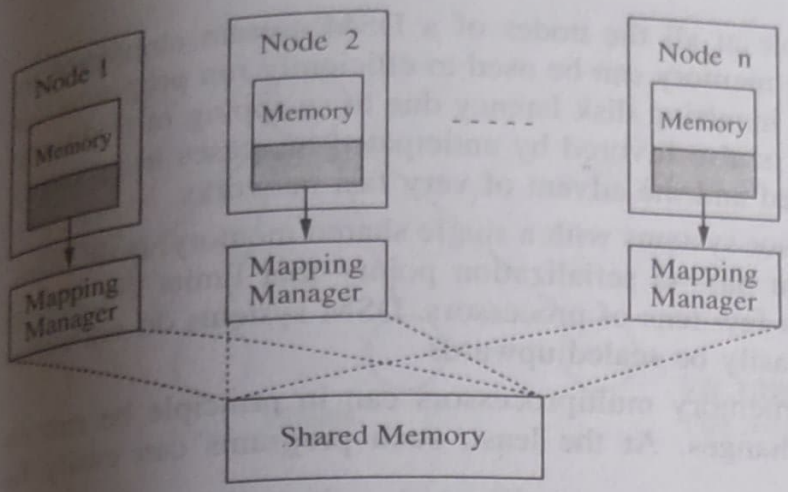

FIGURE 10.1
Distributed shared memory (adapted from [28]).

when data moves from one node to another. When a process accesses data in the shared address space, a *mapping manager* maps the shared memory address to the physical memory (which can be local or remote). The mapping manager is a layer of software implemented either in the operating system kernel or as a runtime library routine. To reduce delays due to communication latency, DSM may move data at the shared memory address from a remote node to the node that is accessing data (when the shared memory address maps to a physical memory location on a remote node). In such cases, DSM makes use of the communication services of the underlying communication system.

Advantages of Distributed Shared Memory are:

1. In the message passing model, programs make shared data available through explicit message passing. In other words, programmers need to be conscious of the data movement between processes. Programmers have to explicitly use communication primitives (such as SEND and RECEIVE), a task that places a significant burden on them. In contrast, DSM systems hide this explicit data movement and provide a simpler abstraction for sharing data that programmers are already well versed with. Hence, it is easier to design and write parallel algorithms using DSM rather than through explicit message passing.
2. In the message passing model, data moves between two different address spaces. This makes it difficult to pass complex data structures between two processes. Moreover, passing data by reference and passing data structures containing pointers is generally difficult and expensive. In contrast, DSM systems allow complex structures to be passed by reference, thus simplifying the development of algorithms for distributed applications.
3. By moving the entire block or page containing the data referenced to the site of reference instead of moving only the specific piece of data referenced, DSM takes advantage of the locality of reference exhibited by programs and thereby cuts down on the overhead of communicating over the network.
4. DSM systems are cheaper to build than tightly coupled multiprocessor systems. This is because DSM systems can be built using off-the-shelf hardware and do not require complex interfaces to connect the shared memory to the processors.

5. The physical memory available at all the nodes of a DSM system combined together is enormous. This large memory can be used to efficiently run programs that require large memory without incurring disk latency due to swapping in traditional distributed systems. This fact is also favored by anticipated increases in processor speed relative to memory speed and the advent of very fast networks.
6. In tightly coupled multiprocessor systems with a single shared memory, main memory is accessed via a common bus—a serialization point—that limits the size of the multiprocessor system to a few tens of processors. DSM systems do not suffer from this drawback and can easily be scaled upwards.
7. Programs written for shared memory multiprocessors can in principle be run on DSM systems without any changes. At the least, such programs can easily be ported to DSM systems.

In essence, DSM systems strive to overcome the architectural limitations of shared memory machines and to reduce the effort required to write parallel programs in distributed systems.

10.3 ALGORITHMS FOR IMPLEMENTING DSM

The central issues in the implementation of DSM are: (a) how to keep track of the location of remote data, (b) how to overcome the communication delays and high overhead associated with the execution of communication protocols in distributed systems when accessing remote data, and (c) how to make shared data concurrently accessible at several nodes in order to improve system performance. We now describe four basic algorithms to implement DSM systems [31].

10.3.1 The Central-Server Algorithm

In the central-server algorithm [31], a central-server maintains all the shared data. It services the read requests from other nodes or clients by returning the data items to them (see Fig. 10.2). It updates the data on write requests by clients and returns acknowledgment messages. A timeout can be employed to resend the requests in case of failed acknowledgments. Duplicate write requests can be detected by associating sequence numbers with write requests. A failure condition is returned to the application trying to access shared data after several retransmissions without a response.

While the central-server algorithm is simple to implement, the central-server can become a bottleneck. To overcome this problem, shared data can be distributed among several servers. In such a case, clients must be able to locate the appropriate server for every data access. Multicasting data access requests is undesirable as it does not reduce the load at the servers compared to the central-server scheme. A better way to distribute data is to partition the shared data by address and use a mapping function to locate the appropriate server.

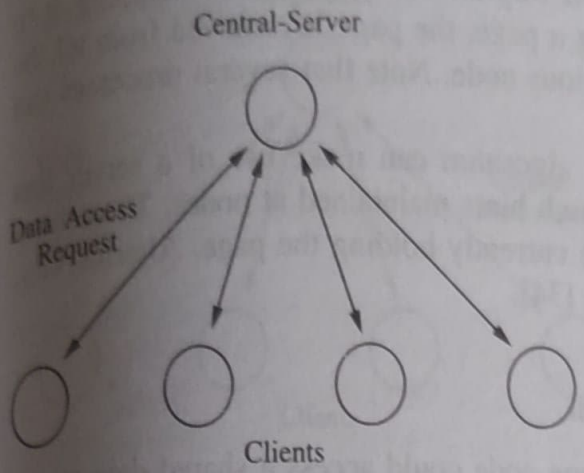

FIGURE 10.2
The central-server algorithm (adapted from [31]).

10.3.2 The Migration Algorithm

In contrast to the central-server algorithm, where every data access request is forwarded to the location of data, data in the migration algorithm is shipped to the location of the data access request, allowing subsequent accesses to the data to be performed locally [31] (see Fig. 10.3). The migration algorithm allows only one node to access a shared data at time.

Typically, the whole page or block containing the data item migrates instead of an individual item requested. This algorithm takes advantage of the locality of reference exhibited by programs by amortizing the cost of migration over multiple accesses to the migrated data. However, this approach is susceptible to *thrashing*, where pages frequently migrate between nodes while servicing only a few requests.

To reduce thrashing, the Mirage system [18] uses a tunable parameter that determines the duration for which a node can possess a shared data item. This allows a node to make a number of accesses to the page before it is migrated to another node. The Munin system [5] strives to reduce data movement by employing protocols that are appropriate to different data access patterns (see Sec. 10.5.3 for details).

The migration algorithm provides an opportunity to integrate DSM with the virtual memory provided by the operating system running at individual nodes. When the page size used by DSM is a multiple of the virtual memory page size, a locally held shared memory page can be mapped to an application's virtual address space and accessed using normal machine instructions. On a memory access fault, if the memory address

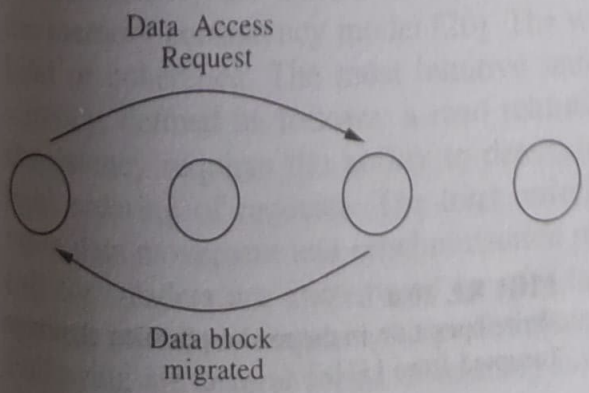

FIGURE 10.3
The migration algorithm (adapted from [31]).

maps to a remote page, a fault-handler will migrate the page before mapping it to the process's address space. Upon migrating a page, the page is removed from all the address spaces it was mapped to at the previous node. Note that several processes can share a page at a node.

To locate a data block, the migration algorithm can make use of a server that keeps track of the location of pages, or through hints maintained at nodes. These hints direct the search for a page toward the node currently holding the page. Alternatively, a query can be broadcasted to locate a page [34].

10.3.3 The Read-Replication Algorithm

In previous approaches, only processes on one node could access a shared data at any one moment. The read-replication algorithm [31] extends the migration algorithm by replicating data blocks and allowing multiple nodes to have read access or one node to have read-write access (the multiple readers-one writer protocol). Read-replication can improve system performance by allowing multiple nodes to access data concurrently. However, the write operation is expensive as all the copies of a shared block at various nodes will either have to be invalidated (see Fig. 10.4) or updated with the current value to maintain the consistency of the shared data block.

In the read-replication algorithm, DSM must keep track of the location of all the copies of data blocks. In the IVY system [27], the owner node of a data block keeps track of all the nodes that have a copy of the data block. In the PLUS system [8], a distributed linked-list is used to keep track of all the nodes that have a copy of the data block.

Nevertheless, read-replication has the potential to reduce the average cost of read operations when the ratio of reads to writes is large. Many read-replication algorithms implemented in the IVY system are described in Sec. 10.7.1.

10.3.4 The Full-Replication Algorithm

The full-replication algorithm [31] is an extension of the read-replication algorithm. It allows multiple nodes to have both read and write access to shared data blocks (the multiple readers-multiple writers protocol). Because many nodes can write shared data concurrently, the access to shared data must be controlled to maintain its consistency.

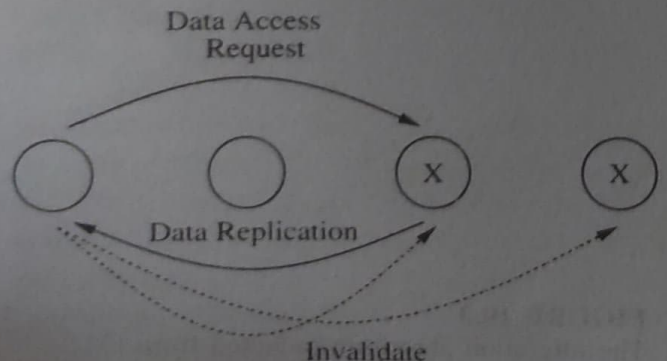

FIGURE 10.4
Write operation in the read-replication algorithm (adapted from [31]).

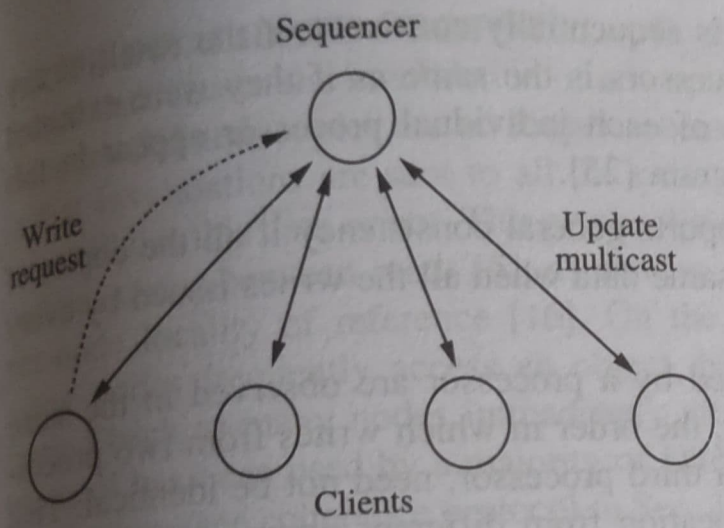

FIGURE 10.5
Write operation in the full-replication algorithm (adapted from [31]).

One simple way to maintain consistency is to use a gap-free sequencer [31]. In this scheme, all nodes wishing to modify shared data will send the modifications to a sequencer. The sequencer will assign a sequence number and multicast the modification with the sequence number to all the nodes that have a copy of the shared data item (see Fig. 10.5). Each node processes the modification requests in the sequence number order. A gap between the sequence number of a modification request and the expected sequence number at a node indicates that one or more modifications have been missed. Under such circumstances, the node will ask for the retransmission of the modifications it has missed. (This implies that a log of the modifications is kept at some node.) Several other protocols to maintain consistency of shared data are discussed in Sec. 10.5.

10.4 MEMORY COHERENCE

DSM systems rely on replicating shared data items and al- if the concurrent accesses are not

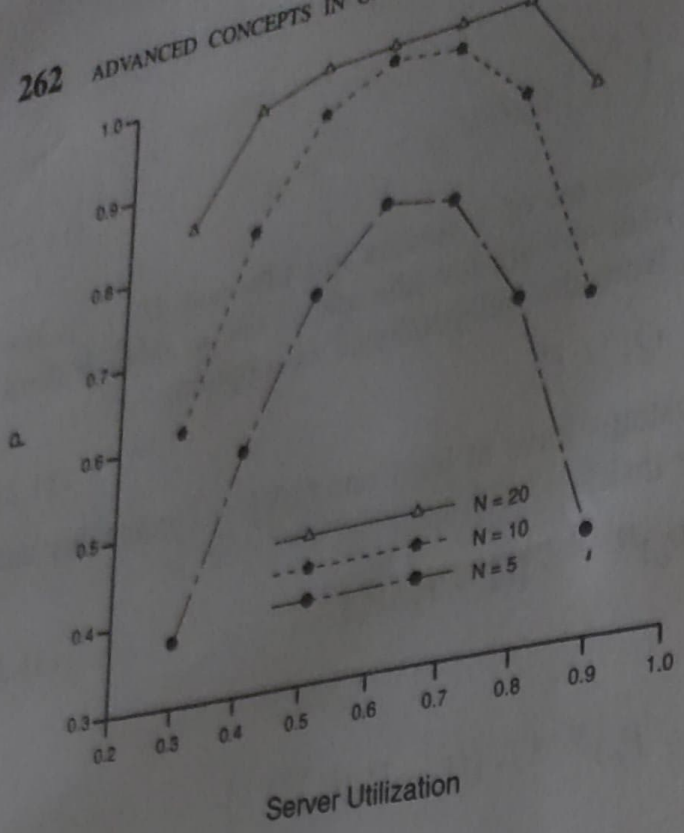

FIGURE 11.2
P as a function of ρ and N (adapted from [24]).

11.3 ISSUES IN LOAD DISTRIBUTING

We now discuss several central issues in load distributing that will help the reader understand its intricacies. Note here that the terms computer, machine, host, workstation, and node are used interchangeably, depending upon the context.

11.3.1 Load

Zhou [41] showed that resource queue lengths and particularly the CPU queue length are good indicators of load because they correlate well with the task response time. Moreover, measuring the CPU queue length is fairly simple and carries little overhead. If a task transfer involves significant delays, however, simply using the current CPU queue length as a load indicator can result in a node accepting tasks while other tasks it accepted earlier are still in transit. As a result, when all the tasks that the node has accepted have arrived, the node can become overloaded and require further task transfers to reduce its load. This undesirable situation can be prevented by artificially incrementing the CPU queue length at a node whenever the node accepts a remote task. To avoid anomalies when task transfers fail, a timeout (set at the time of acceptance) can be employed. After the timeout, if the task has not yet arrived, the CPU queue length is decremented.

While the CPU queue length has been extensively used in previous studies as a load indicator, it has been reported that little correlation exists between CPU queue length and processor utilization [35], particularly in an interactive environment. Hence, the designers of V-System used CPU utilization as an indicator of the load at a site. This approach requires a background process that monitors CPU utilization continuously and imposes more overhead, compared to simply finding the queue length at a node (see Sec. 11.10.1).

11.3.2 Classification of Load Distributing Algorithms

The basic function of a load distributing algorithm is to transfer load (tasks) from heavily loaded computers to idle or lightly loaded computers. Load distributing algorithms can be broadly characterized as *static*, *dynamic*, or *adaptive*. Dynamic load distributing algorithms [3, 10, 11, 18, 20, 24, 31, 34, 40] use system state information (the loads at nodes), at least in part, to make load distributing decisions, while static algorithms make no use of such information. In static load distributing algorithms, decisions are hard-wired in the algorithm using a priori knowledge of the system. Dynamic load distributing algorithms have the potential to outperform static load distributing algorithms because they are able to exploit short term fluctuations in the system state to improve performance. However, dynamic load distributing algorithms entail overhead in the collection, storage, and analysis of system state information. Adaptive load distributing algorithms [20, 31] are a special class of dynamic load distributing algorithms in that they adapt their activities by dynamically changing the parameters of the algorithm to suit the changing system state. For example, a dynamic algorithm may continue to collect the system state irrespective of the system load. An adaptive algorithm, on the other hand, may discontinue the collection of the system state if the overall system load is high to avoid imposing additional overhead on the system. At such loads, all nodes are likely to be busy and attempts to find receivers are unlikely to be successful.

11.3.3 Load Balancing vs. Load Sharing

Load distributing algorithms can further be classified as *load balancing* or *load sharing* algorithms, based on their load distributing principle. Both types of algorithms strive to reduce the likelihood of an *unshared state* (a state in which one computer lies idle while at the same time tasks contend for service at another computer [21]) by transferring tasks to lightly loaded nodes. Load balancing algorithms [7, 20, 24], however, go a step further by attempting to equalize loads at all computers. Because a load balancing algorithm transfers tasks at a higher rate than a load sharing algorithm, the higher overhead incurred by the load balancing algorithm may outweigh this potential performance improvement.

Task transfers are not instantaneous because of communication delays and delays that occur during the collection of task state. Delays in transferring a task increase the duration of an unshared state as an idle computer must wait for the arrival of the transferred task. To avoid lengthy unshared states, *anticipatory* task transfers from overloaded computers to computers that are likely to become idle shortly can be used. Anticipatory transfers increase the task transfer rate of a load sharing algorithm, making it less distinguishable from load balancing algorithms. In this sense, load balancing can be considered a special case of load sharing, performing a particular level of anticipatory task transfers.

11.3.4 Preemptive vs. Nonpreemptive Transfers

Preemptive task transfers involve the transfer of a task that is partially executed. This transfer is an expensive operation as the collection of a task's state (which can be quite

large and complex) can be difficult. Typically, a task state consists of a virtual memory image, a process control block, unread I/O buffers and messages, file pointers, timers that have been set, etc. Nonpreemptive task transfers, on the other hand, involve the transfer of tasks that have not begun execution and hence do not require the transfer of the task's state. In both types of transfers, information about the environment in which the task will execute must be transferred to the receiving node. This information can include the user's current working directory, the privileges inherited by the task, etc. Nonpreemptive task transfers are also referred to as *task placements*.

11.4 COMPONENTS OF A LOAD DISTRIBUTING ALGORITHM

Typically, a load distributing algorithm has four components: (1) a *transfer* policy that determines whether a node is in a suitable state to participate in a task transfer, (2) a *selection* policy that determines *which* task should be transferred, (3) a *location* policy that determines to which node a task selected for transfer should be sent, and (4) an *information policy* which is responsible for triggering the collection of system state information. A transfer policy typically requires information on the local node's state to make decisions. A location policy, on the other hand, is likely to require information on the states of remote nodes to make decisions.

11.4.1 Transfer Policy

A large number of the transfer policies that have been proposed are *threshold* policies [10, 11, 24, 31]. Thresholds are expressed in units of load. When a new task originates at a node, and the load at that node exceeds a threshold T, the transfer policy decides that the node is a *sender*. If the load at a node falls below T, the transfer policy decides that the node can be a *receiver* for a remote task.

An alternative transfer policy initiates task transfers whenever an imbalance in load among nodes is detected because of the actions of the information policy.

11.4.2 Selection Policy

A selection policy selects a task for transfer, once the transfer policy decides that the node is a sender. Should the selection policy fail to find a suitable task to transfer, the node is no longer considered a sender until the transfer policy decides that the node is a sender again.

The simplest approach is to select newly originated tasks that have caused the node to become a sender by increasing the load at the node beyond the threshold [11]. Such tasks are relatively cheap to transfer, as the transfer is nonpreemptive.

A basic criterion that a task selected for transfer should satisfy is that the overhead incurred in the transfer of the task should be compensated for by the reduction in the response time realized by the task. In general, long-lived tasks satisfy this criterion [4]. Also, a task can be selected for remote execution if the estimated average execution time for that type of task is greater than some execution time threshold [36].

Bryant and Finkel [3] propose another approach based on the reduction in response time that can be obtained for a task by transferring it elsewhere. In this method, a task is selected for transfer only if its response time will be improved upon transfer. (See [3] for details on how to estimate response time.)

There are other factors to consider in the selection of a task. First, the overhead incurred by the transfer should be minimal. For example, a task of small size carries less overhead. Second, the number of location-dependent system calls made by the selected task should be minimal. Location-dependent calls must be executed at the node where the task originated because they use resources such as windows, or the mouse, that only exist at the node [8, 19].

11.4.3 Location Policy

The responsibility of a location policy is to find suitable nodes (senders or receivers) to share load. A widely used method for finding a suitable node is through *polling*. In polling, a node polls another node to find out whether it is a suitable node for load sharing [3, 10, 11, 24, 31]. Nodes can be polled either serially or in parallel (e.g., multicast). A node can be selected for polling either randomly [3, 10, 11], based on the information collected during the previous polls [24, 31], or on a nearest-neighbor basis. An alternative to polling is to broadcast a query to find out if any node is available for load sharing.

11.4.4 Information Policy

The information policy is responsible for deciding when information about the states of other nodes in the system should be collected, where it should be collected from, and what information should be collected. Most information policies are one of the following three types:

Demand-driven. In this class of policy, a node collects the state of other nodes only when it becomes either a sender or a receiver (decided by the transfer and selection policies at the node), making it a suitable candidate to initiate load sharing. Note that a demand-driven information policy is inherently a dynamic policy, as its actions depend on the system state. Demand-driven policies can be *sender-initiated*, *receiver-initiated*, or *symmetrically initiated*. In sender-initiated policies, senders look for receivers to transfer their load. In receiver-initiated policies, receivers solicit load from senders. A symmetrically initiated policy is a combination of both, where load sharing actions are triggered by the demand for extra processing power or extra work.

Periodic. In this class of policy, nodes exchange load information periodically [14, 40]. Based on the information collected, the transfer policy at a node may decide to transfer jobs. Periodic information policies do not adapt their activity to the system state. For example, the benefits due to load distributing are minimal at high system loads because most of the nodes in the system are busy. Nevertheless, overheads due to periodic information collection continue to increase the system load and thus worsen the situation.

State-change-driven. In this class of policy, nodes disseminate state information whenever their state changes by a certain degree [24]. A state-change-driven policy differs from a demand-driven policy in that it disseminates information about the state of a node, rather than collecting information about other nodes. Under centralized state-change-driven policies, nodes send state information to a centralized collection point. Under decentralized state-change-driven policies, nodes send information to peers [32].

11.5 STABILITY

We now describe two views of stability.

11.5.1 The Queuing-Theoretic Perspective

When the long term arrival rate of work to a system is greater than the rate at which the system can perform work, the CPU queues grow without bound. Such a system is termed unstable. For example, consider a load distributing algorithm performing excessive message exchanges to collect state information. The sum of the load due to the external work arriving and the load due to the overhead imposed by the algorithm can become higher than the service capacity of the system, causing system instability.

Alternatively, an algorithm can be stable but may still cause a system to perform worse than when it is not using the algorithm. Hence, a more restrictive criterion for evaluating algorithms is desirable, and we use the *effectiveness* of an algorithm as the evaluating criterion. A load distributing algorithm is said to be effective under a given set of conditions if it improves the performance relative to that of a system not using load distributing. Note that while an effective algorithm cannot be unstable, a stable algorithm can be ineffective.

11.5.2 The Algorithmic Perspective

If an algorithm can perform fruitless actions indefinitely with finite probability, the algorithm is said to be unstable [3]. For example, consider *processor thrashing*. The transfer of a task to a receiver may increase the receiver's queue length to the point of overload, necessitating the transfer of that task to yet another node. This process may repeat indefinitely [3]. In this case, a task is moved from one node to another in search of a lightly loaded node without ever receiving service. Discussions on various types of algorithmic instability are beyond the scope of this book and can be found in [6].

11.6 LOAD DISTRIBUTING ALGORITHMS

We now describe some load distributing algorithms that have appeared in the literature and discuss their performance.

11.6.1 Sender-Initiated Algorithms

In sender-initiated algorithms, load distributing activity is initiated by an overloaded node (sender) that attempts to send a task to an underloaded node (receiver). This section covers three simple yet effective sender-initiated algorithms studied by Eager, Lazowska, and Zohorjan [11].

Transfer policy. All three algorithms use the same transfer policy, a threshold policy based on CPU queue length. A node is identified as a sender if a new task originating at the node makes the queue length exceed a threshold T. A node identifies itself as a suitable receiver for a remote task if accepting the task will not cause the node's queue length to exceed T.

Selection policy. These sender-initiated algorithms consider only newly arrived tasks for transfer.

Location policy. These algorithms differ only in their location policy:

Random. Random is a simple dynamic location policy that uses no remote state information. A task is simply transferred to a node selected at random, with no information exchange between the nodes to aid in decision making. A problem with this approach is that useless task transfers can occur when a task is transferred to a node that is already heavily loaded (i.e., its queue length is above the threshold). An issue raised with this policy concerns the question of how a node should treat a transferred task. If it is treated as a new arrival, the transferred task can again be transferred to another node if the local queue length is above the threshold. Eager et al. [11] have shown that if such is the case, then irrespective of the average load of the system, the system will eventually enter a state in which the nodes are spending all their time transferring tasks and not executing them. A simple solution to this problem is to limit the number of times a task can be transferred. A sender-initiated algorithm using the random location policy provides substantial performance improvement over no load sharing at all [11].

Threshold. The problem of useless task transfers under random location policy can be avoided by polling a node (selected at random) to determine whether it is a receiver (see Fig. 11.3). If so, the task is transferred to the selected node, which must execute the task regardless of its state when the task actually arrives. Otherwise, another node is selected at random and polled. The number of polls is limited by a parameter called *PollLimit* to keep the overhead low. Note that while nodes are randomly selected, a sender node will not poll any node more than once during one searching session of PollLimit polls. If no suitable receiver node is found within the PollLimit polls, then the node at which the task originated must execute the task. By avoiding useless task transfers, the threshold policy provides substantial performance improvement over the random location policy [11].

Shortest. The two previous approaches make no effort to choose the best receiver for a task. Under the *shortest* location policy, a number of nodes (= PollLimit) are

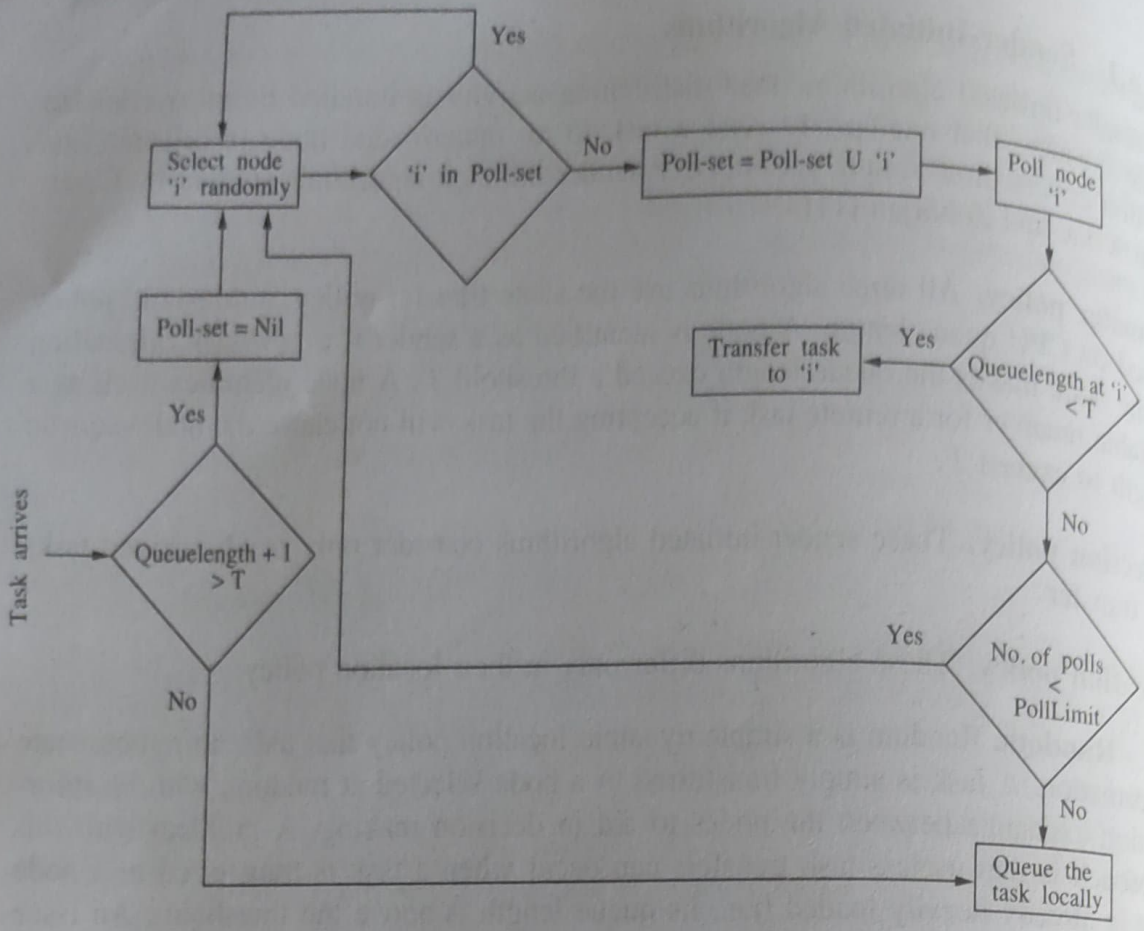

FIGURE 11.3
Sender-initiated load sharing with threshold location policy.

selected at random and are polled to determine their queue length [11]. The node with the shortest queue length is selected as the destination for task transfer unless its queue length $\geq T$. The destination node will execute the task regardless of its queue length at the time of arrival of the transferred task. The performance improvement obtained by using the shortest location policy over the threshold policy was found to be marginal [11], indicating that using more detailed state information does not necessarily result in significant improvement in system performance.

Information policy. When either the shortest or the threshold location policy is used, polling activity commences when the transfer policy identifies a node as the sender of a task. Hence, the information policy can be considered to be of the demand-driven type.

Stability. These three approaches for location policy used in sender-initiated algorithms cause system instability at high system loads, where no node is likely to be lightly loaded, and hence the probability that a sender will succeed in finding a receiver node is very low. However, the polling activity in sender-initiated algorithms increases as the rate at which work arrives in the system increases, eventually reaching a point where the cost of load sharing is greater than the benefit. At this point, most of the available CPU cycles are wasted in unsuccessful polls and in responding to these polls. When the load due to work arriving and due to the load sharing activity exceeds the system's

serving capacity, instability occurs. Thus, the actions of sender-initiated algorithms are not effective at high system loads and cause system instability by failing to adapt to the system state.

11.6.2 Receiver-Initiated Algorithms

In receiver-initiated algorithms, the load distributing activity is initiated from an underloaded node (receiver) that is trying to obtain a task from an overloaded node (sender). In this section, we describe the policies of an algorithm [31] that is a variant of the algorithm proposed in [10] (see Fig. 11.4).

Transfer policy. Transfer policy is a threshold policy where the decision is based on CPU queue length. The transfer policy is triggered when a task departs. If the local queue length falls below the threshold T, the node is identified as a receiver for obtaining a task from a node (sender) to be determined by the location policy. A node is identified to be a sender if its queue length exceeds the threshold T.

Selection policy. This algorithm can make use of any of the approaches discussed under the selection policy in Sec. 11.4.2.

Location policy. In this policy, a node selected at random is polled to determine if transferring a task from it would place its queue length below the threshold level. If

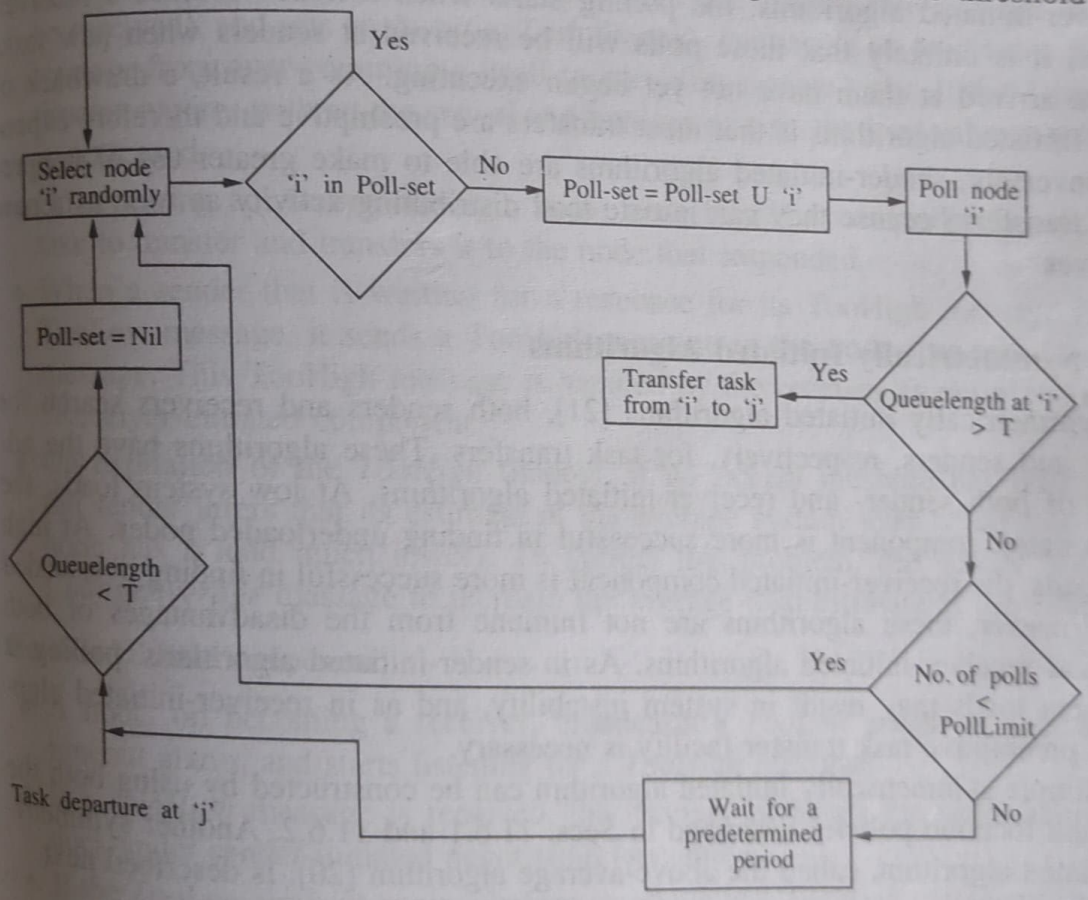

FIGURE 11.4
Receiver-initiated load sharing.

not, the polled node transfers a task. Otherwise, another node is selected at random and the above procedure is repeated until a node that can transfer a task (i.e., a sender) is found, or a static PollLimit number of tries have failed to find a sender. If all polls fail to find a sender, the node waits until another task completes or until a predetermined period is over before initiating the search for a sender, provided the node is still a receiver. Note that if the search does not start after a predetermined period, the extra processing power available at a receiver is completely lost to the system until another task completes, which may not occur soon.

Information policy. The information policy is demand-driven because the polling activity starts only after a node becomes a receiver.

Stability. Receiver-initiated algorithms do not cause system instability for the following reason. At high system loads there is a high probability that a receiver will find a suitable sender to share the load within a few polls. This results in the effective usage of polls from receivers and very little wastage of CPU cycles at high system loads. At low system loads, there are few senders but more receiver-initiated polls. These polls do not cause system instability as spare CPU cycles are available at low system loads.

A drawback. Under the most widely used CPU scheduling disciplines (such as round-robin and its variants), a newly arrived task is quickly provided a quantum of service. In receiver-initiated algorithms, the polling starts when a node becomes a receiver. However, it is unlikely that these polls will be received at senders when new tasks that have arrived at them have not yet begun executing. As a result, a drawback of receiver-initiated algorithms is that most transfers are preemptive and therefore expensive. Conversely, sender-initiated algorithms are able to make greater use of nonpreemptive transfers because they can initiate load distributing activity as soon as a new task arrives.