DISTRIBUTED COMPUTING

John A. Stankovic

COINS Technical Report 92-11
February 1992

DISTRIBUTED COMPUTING[1]

Prof. John A. Stankovic
Department of Computer Science
University of Massachusetts
Amherst

1 Introduction

A distributed computer system (DCS) is a collection of computers connected by a communications subnet and logically integrated in varying degrees by a distributed operating system and/or distributed database system. Each computer node may be a uniprocessor, or multiprocessor, or multicomputer. The communications subnet may be a widely geographically dispersed collection of communication processors or a local area network. Typical applications that use distributed computing include e-mail, teleconferencing, electronic funds transfers, multi-media telecommunications, command and control systems, and support for general purpose computing in industrial and academic settings. The widespread use of distributed computer systems is due to the price-performance revolution in microelectronics, the development of cost effective and efficient communication subnets [4] (which is itself due to the merging of data communications and computer communications), the development of resource sharing software, and the increased user demands for communication, economical sharing of resources, and productivity.

A DCS potentially provides significant advantages, including good performance, good reliability, good resource sharing, and extensibility [35, 41]. Potential performance enhancement is due to multiple processors and an efficient subnet, as well as avoiding contention and bottlenecks that exist in uniprocessors and multiprocessors. Potential reliability improvements are due to the data and control redundancy possible, the geographical distribution of the system, and the ability for hosts and communication processors to perform mutual inspection. With the proper subnet, distributed operating system [46], and distributed database [85], it is possible to share hardware and software resources in a cost effective manner, increasing productivity and lowering costs. Possibly the most important potential advantage of a DCS is extensibility. Extensibility is the ability to easily adapt to both short and long term changes without significant disruption of the system. Short term changes include varying workloads and host or subnet failures or additions. Long term changes are associated with major modifications to the requirements or content of the system.

[1]Invited paper. To appear in *Encyclopedia of Telecommunications*, Marcel Dekker, Inc., N.Y.

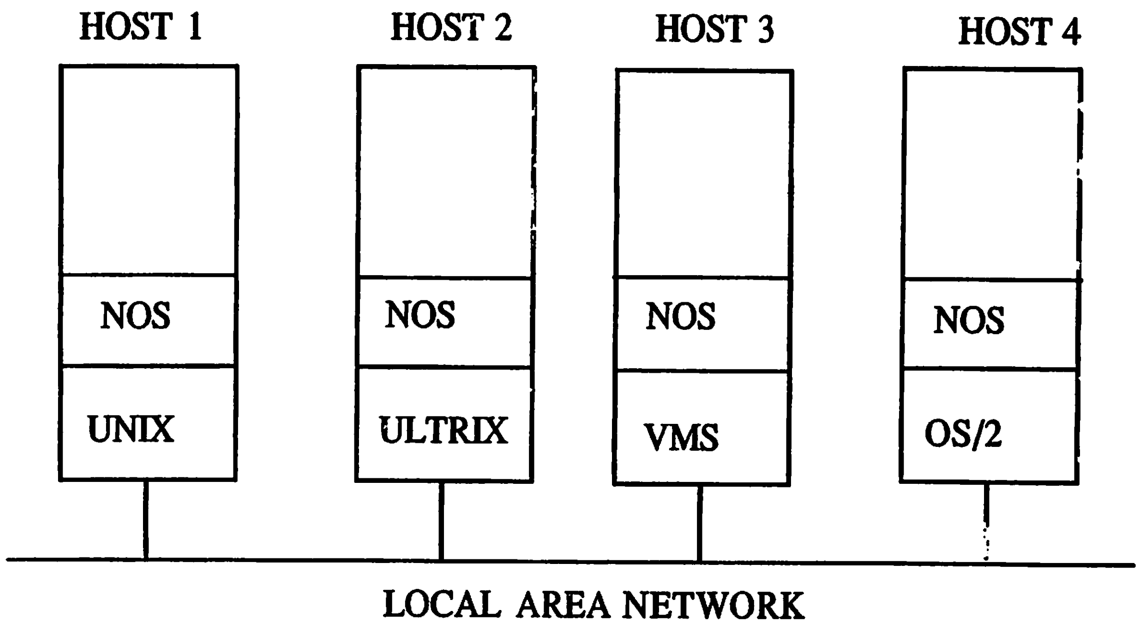
DCS research encompasses many areas, including: local and wide area networks, distributed operating systems, distributed databases, distributed file servers, concurrent and distributed programming languages, specification languages for concurrent system, theory of parallel algorithms, theory of distributed computing, parallel architectures and interconnection structures, fault tolerant and ultrareliable systems, distributed real-time systems, cooperative problem solving techniques of artificial intelligence, distributed debugging, distributed simulation, distributed applications, and a methodology for the design, construction and maintenance of large, complex distributed systems. Many prototype distributed computer systems have been built at university, industrial, commercial, and government research laboratories, and production systems of all sizes and types have proliferated. It is impossible to survey all distributed computing system research. An extensive survey and bibliography would require hundreds of pages. Instead, this chapter focuses on two important areas: distributed operating systems and distributed databases.

2 Distributed Operating Systems

Operating systems for distributed computing systems can be categorized into two broad categories: network operating systems and distributed operating systems [113].

Network Operating Systems: Consider the situation where each of the hosts of a computer network has a local operating system that is independent of the network. The sum total of all the operating system software added to each host in order to communicate and share resources is called a network operating system (NOS). The added software often includes modifications to the local operating system. NOS's are characterized by being built on top of existing operating systems, and they attempt to hide the differences between the underlying systems. See Figure 1.

Distributed Operating Systems: Consider an integrated computer network where there is one native operating system for all the distributed hosts. This is called a distributed operating system (DOS). See Figure 2. Examples of DOSs include the V system [28], Eden [66], Amoeba [78], the Cambridge distributed computing system [79], Medusa [83], Locus [87], and Mach [93]. Examples of real-time distributed operating systems include MARS [59] and Spring [116]. A DOS is designed with the network requirements in mind from its inception and it tries to manage the resources of the network in a global fashion. Therefore, retrofitting a DOS to existing operating systems and other software is not a problem for DOSs. Since DOSs are used to satisfy a wide variety of requirements, their various implementations are quite different. Note that the boundary between NOS's and DOSs is not always clearly distinguishable. In this chapter we primarily consider distributed operating systems issues divided into six categories: process structures, access control and communication, reliability, heterogeneity, efficiency, and real-time. In section 3 we present a similar breakdown applied



NATIVE OPERATING SYSTEMS - UNIX,ULTRIX,VMS,OS/2

FIGURE 1: NETWORK OPERATING SYSTEMS (NOS)

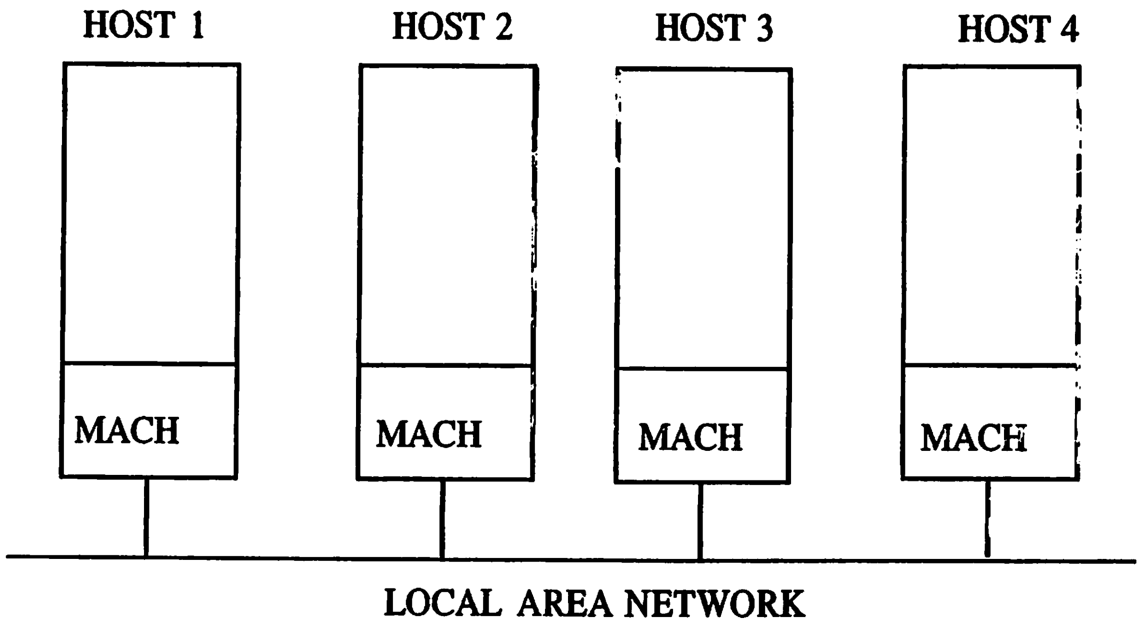


FIGURE 2: DISTRIBUTED OPERATING SYSTEM - MACH

to distributed databases.

2.1 Process Structures

The conventional notion of a process is an address space with a single execution trace through it. Because of the parallelism inherent in multiprocessing and distributed computing, we have seen that recent operating systems are supporting the separation of address space (sometimes called a task) and execution traces (called threads or lightweight processes) [93, 94]. In most systems the address space and threads are restricted to reside on a single node (a uni- or multi-processor). However, some systems such as Ivy, the Apollo domain, and Clouds [33] support a distributed address space (sometimes called a distributed shared memory – see [82] for a summary of issues involved with distributed shared memory), and distributed threads executing on that address space. Regardlesss of whether the address space is local or distributed [42], there has been significant work done the following topics. They are: supporting very large, but sparse address spaces, efficiently copying information between address spaces using a technique called *copy on write* where only the data actually used gets copied [43], and supporting efficient file management by mapping files into the address space and then using virtual memory techniques to access the file.

At a higher level distributed operating systems use tasks and threads to support either a procedure or an object based paradigm [29]. If objects are used, there are two variations: the passive and active object models. Because the object based paradigm is so important and well suited to distributed computing, we will present some basic information about objects and then discuss the active and passive object paradigms.

A data abstraction is a collection of information and a set of operations defined on that information. An object is an instantiation of a data abstraction. The concept of an object is usually supported by a kernel which may also define a primitive set of objects. Higher level objects are then constructed from more primitive objects in some structured fashion. All hardware and software resources of a DCS can be regarded as objects. The concept of an object and its implications form an elegant basis for a DCS [83, 113]. For example, distributed systems functions such as allocation of objects to a host, moving objects, remote access to objects, sharing objects across the network, and providing interfaces between disparate objects are all "conceptually" simple, because they are all handled by yet other objects. The object concept is powerful and can easily support the popular client-server model of distributed computing.

Objects also serve as the primitive entity supporting more complicated distributed computational structures. One type of distributed computation is a process (thread) which executes as a sequential trace through passive objects, but does so across multiple hosts. See Figure 3. The objects are permanent but the execution properties are supplied by an external process (thread) executing in the address space of the object. Another form of a distributed compu-

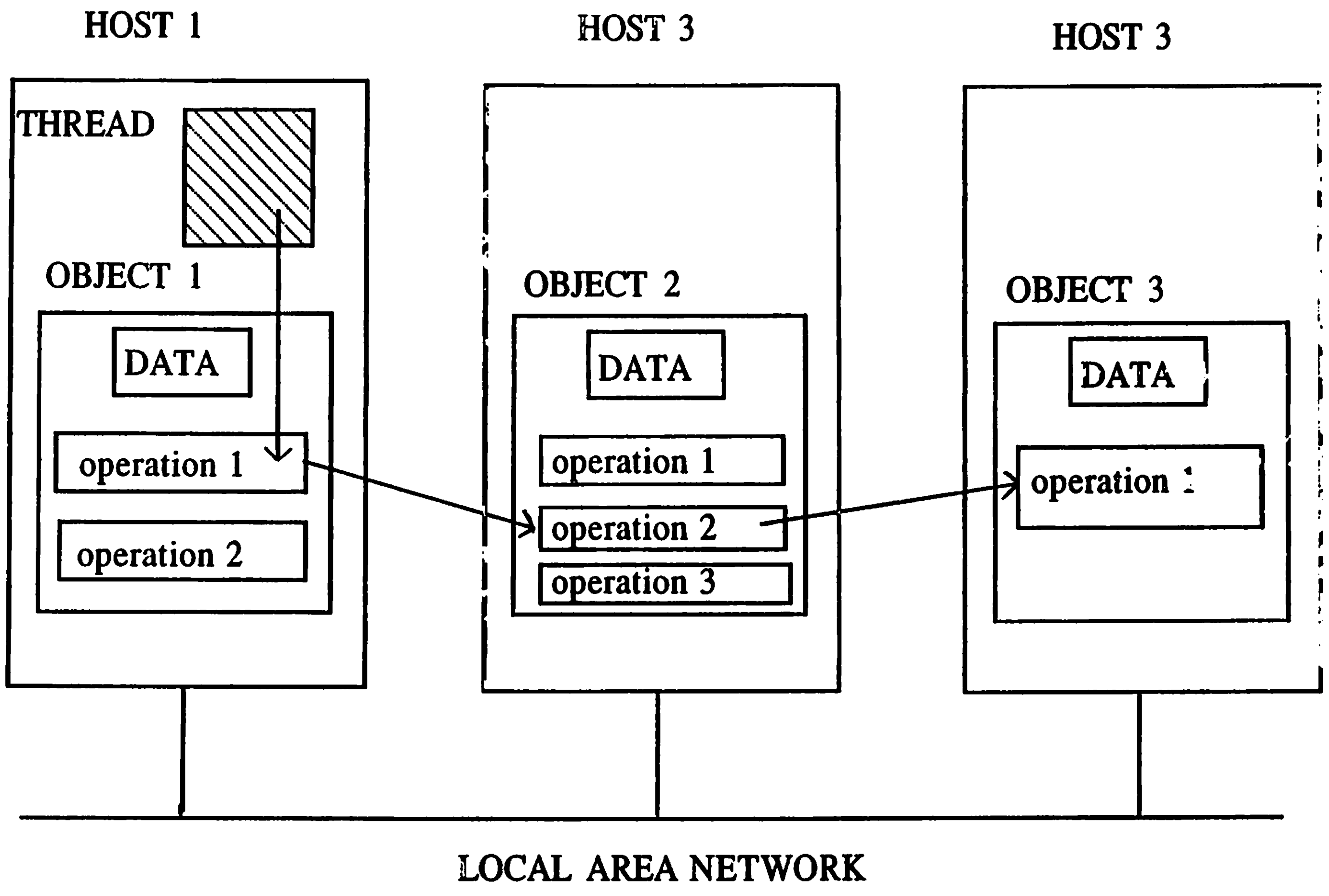


FIGURE 3: DISTRIBUTED COMPUTATION (THREAD THROUGH PASSIVE OBJECTS).

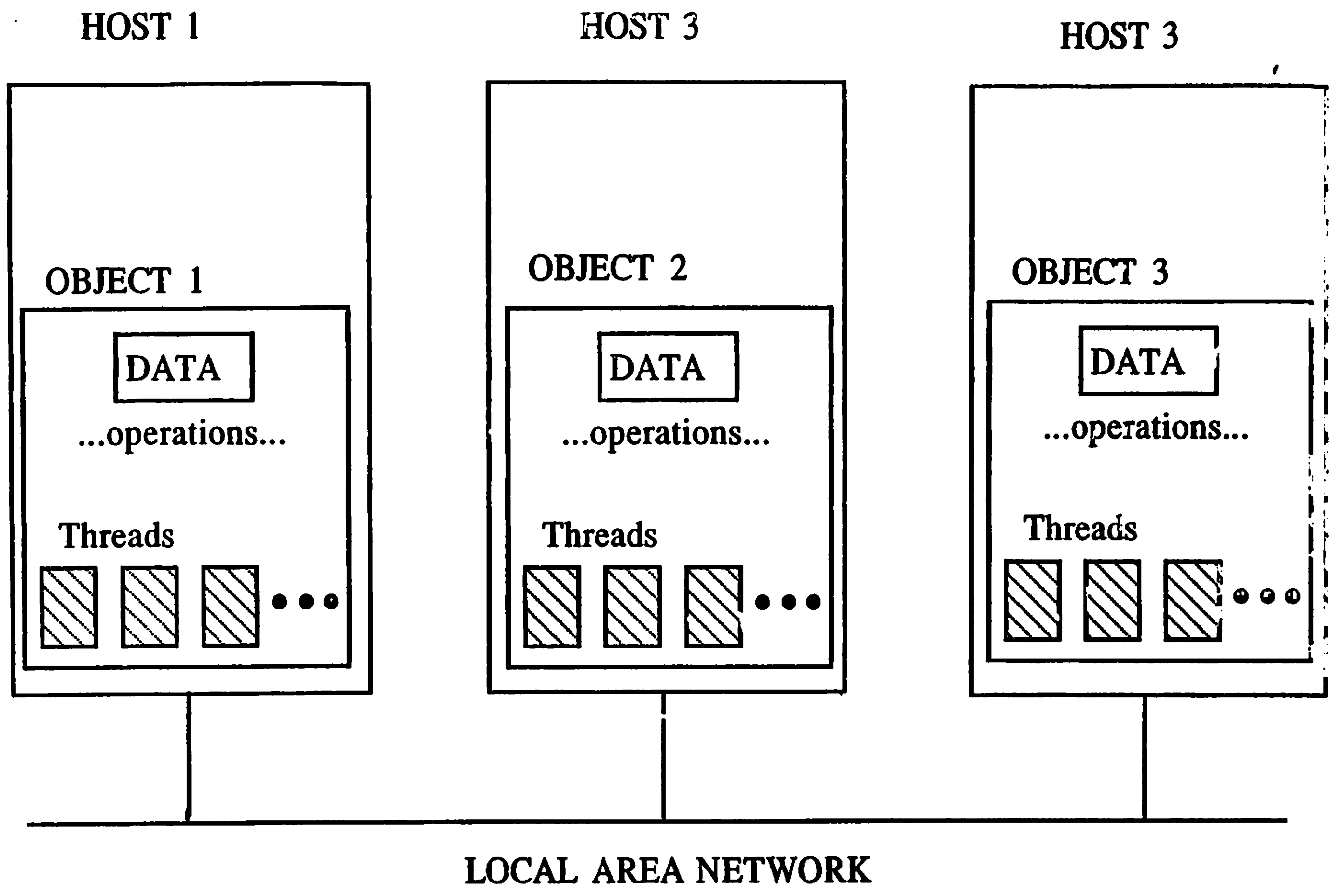


FIGURE 4: ACTIVE OBJECTS

tation is to have clusters of objects, each with internal, active threads, running in parallel and communicating with each other based on the various types of interprocess communication (IPC) used. This is known as the active object model. See Figure 4. The cluster of processes may be co-located or distributed in some fashion. Other notions of what constitutes a distributed program are possible, e.g., object invocations can support additional semantics such as found in database transactions.

The major problem with object based systems has been poor execution time performance. However, this is not really a problem with the object abstraction itself, but is a problem with inefficient implementation of access to objects. The most common reason given for poor execution time is that current architectures are ill suited for object based systems. Another problem is choosing the right granularity for an object. If every integer or character and their associated operations are treated as objects, then the overhead is too high. If the granularity of an object is too large, then the benefits of the object based system are lost.

2.2 Access Control and Communications

A distributed system consists of a collection of resources managed by the distributed operating system. Accessing the resources must be controlled in two ways. First, the manner used to access the resource must be suitable to the resource and requirements under consideration. For example, a printer must be shared serially, local data of an object should not be shareable, and a read only file can be accessed simultaneously by any number of users. In an object based system access can be controlled on an operation by operation basis. For example, a given user may be restricted to only using the insert operation on a queue object while another user may be able to access the queue using both insert and remove operations. Second, access to a resource must be restricted to a set of allowable users. In many systems this is done by an access control list, an access control matrix, or capabilities. The MIT Athena project developed an authentication server called Kerberos based on a third party authentication model [80] that uses private key encryption.

Communication has been the focus of much of the work in distributed operating systems providing the glue that binds logically and physically separated processes [54, 92]. Remote procedure calls (RPC) extend the semantics of programming language's procedure calls to communication across nodes of a DCS. Lightweight RPCs [18] have been developed to make such calls as efficient as possible. Many systems also support general synchronous and asynchronous send and receive primitives whose semantics are different and more general than RPC semantics. Broadcasting and multicasting are also common primitives found in DOSs and they provide useful services in achieving consensus and other forms of global coordination. For systems with high reliability requirements, reliable broadcast facilities might be provided [55].

Implementing communication facilities is done either directly in the kernels of the operating systems (as in the V system [28]) or as user level services (as in MACH [93]). This is a classical tradeoff between performance and flexibility. Intermediate between these approaches lies the x-kernel [54] where basic primitives required for all communication primitives are provided at the kernel level and protocol specific logic is programmable at a higher level.

2.3 Reliability

While reliability is a fundamental issue for any system, the redundancy found in DCSs make them particularly well suited for the implementation of reliability techniques. We begin the discussion on reliability with a few definitions.

A *fault* is a mechanical or algorithmic defect which may generate an error. A fault may be permanent, transient, or intermittent. An *error* is an item of information which when processed by the normal algorithms of the system will produce a failure. A *failure* is an event at which a system violates its specifications. *Reliability* can then be defined as the degree of tolerance against errors and faults. Increased reliability comes from fault avoidance and fault tolerance. *Fault avoidance* results from conservative design practices such as using high reliability components and non-ambitious design. *Fault tolerance* employs error detection and redundancy to deal with faults and errors. Most of what we discuss in this section relates to the fault tolerance aspect of reliability.

Reliability is a complex, multidimensional activity that must simultaneously address some or all of the following: fault confinement, fault detection, fault masking, retries, fault diagnosis, reconfiguration, recovery, restart, repair, and reintegration. Further, distributed systems require more than reliability, i.e., they need to be *dependable*. Dependability is the trustworthiness of a computer system and it subsumes reliability, availability, safety, and security. System architectures such as Delta-4 [88] strive for dependability. We cannot do justice to all these issues in this short article. Instead we will discuss several of the more important issues related to reliability in DOSs.

Reliable DOSs should support replicated files, exception handlers, testing procedures executed from remote hosts, and avoid single points of failure by a combination of replication, backup facilities, and distributed control. Distributed control could be used for file servers, name servers, scheduling algorithms, and other executive control functions. Process structure, how environment information is kept, the homogeneity of various hosts, and the scheduling algorithm may allow for relocatability of processes. Interprocess communication (IPC) might be supported as a reliable remote procedure call [81, 109] and also provide reliable atomic broadcasts as is done in ISIS [19]. Reliable IPC would enforce "at least once" or "exactly once" semantics depending on the type of IPC being invoked, and atomic broadcasts guarantees that either all processes that are to receive the message will indeed receive it, or none will. Other DOS reliability solutions are required to avoid invoking processes that are not active, to avoid the situation where a process remains active, but is not used, and to avoid attempts to communicate to terminated processes.

ARGUS [70], a distributed programming language, has explicitly incorporated reliability concerns into the programming language. It does this by supporting the idea of an atomic object, transactions, nested actions, reliable remote procedure calls, stable variables, guardians (which are modules that service node failures and synchronize concurrent access to data), exception handlers, periodic and background testing procedures, and recovery of a committed update given the present update does not complete. A distributed program written in ARGUS may potentially experience deadlock. Currently, deadlocks are broken by timing out and aborting actions.

Distributed databases make use of many reliability features such as stable storage, transactions, nested transactions [76], commit and recovery protocols [103], nonblocking commit protocols [102], termination protocols [104], checkpointing, replication, primary/backups, logs/audit trails, differential files [99], and timeouts to detect failures. Operating system support is required to make these mechanisms more efficient [47, 87, 119, 131].

One aspect of reliability not stressed enough in DCS research is the need for robust solutions, i.e., the solutions must explicitly assume an unreliable network, tolerate host failures, network partitionings, and lost, duplicate, out of order, or noisy data. Robust algorithms must sometimes make decisions after reaching only approximate agreement or by using statistical properties of the system (assumed known or dynamically calculated). A related question is at what level should the robust algorithms, and reliability in general, be supported? Most systems attempt to have the subnet ensure reliable, error free data transmission between processes. However, according to the end-to-end argument [97], such functions placed at the lower levels of the system are often redundant and unnecessary. The rationale for this argument is that since the application has to take into account errors introduced not only by the subnet, many of the error detection and recovery functions can be correctly and completely provided only at the application level.

The relationship of reliability to the other issues discussed in this paper is very strong. For example, object based systems confine errors to a large degree, define a consistent system state to support rollback and restart, and limit propagation of rollback activities. However, if objects are supported on a distributed shared memory special problems arise [134]. Since objects can represent unreliable resources (such as processors and disks), and since higher level objects can be built using lower level objects, the goal of reliable system design is to create "reliable" objects out of unreliable objects. For example, a stable storage can be created out of several disk objects and the proper logic. Then a physical processor, a checkpointing capability, a stable storage, and logic can be used to create a stable processor. One can proceed in this fashion to create a very reliable system. The main drawback is potential loss of execution time efficiency. For many systems, it is just too costly to incorporate an extensive number of reliability mechanisms. Reliability is also enhanced by proper access control and judicial use of distributed control. The major challenge is to integrate solutions to all these issues in a cost effective manner and produce an extremely reliable system.

2.4 Heterogeneity

Incompatibility problems arise in heterogeneous DCS's in a number of ways [10] and at all levels. First, incompatibility is due to the different internal formatting schemes that exist in a collection of different communication and host processors. Second, incompatibility also arises from the differences in communication protocols and topology when networks are connected to other networks via gateways. Third, major incompatibilities arise due to different operating systems, file servers, and database systems that might exist on a (set of) network(s).

The easiest solution to this general problem for a single DCS is to avoid the issue by using a homogeneous collection of machines and software. If this is not practical, then some form of translation is necessary. Some earlier systems left this translation to the user. This is no longer acceptable.

Translation done by the DCS system can be done at the receiver host or at the source host. If it is done at the receiver host, then the data traverse the network in their original form. The data usually are supplemented with extra information to guide the translation. The problem with this approach is that at every host there must be a translator to convert each format in the system to the format used on the receiving host. When there exist "n" different formats, this requires the support of $(n - 1)$ translators at each host. Performing the translation at the source host before transmitting the data is subject to all the same problems.

There are two better solutions, each applicable under different situations: an intermediate translator, or an intermediate standard data format.

An intermediate translator accepts data from the source and produces the acceptable format for the destination. This is usually used when the number of different types of necessary conversions is small. For example, a gateway linking two different networks acts as an intermediate translator.

For a given conversion problem, if the number of different types to be dealt with grows large, then a single intermediate translator becomes unmanageable. In this case, an intermediate standard data format (interface) is declared, hosts convert to the standard, data are moved in the format of the standard, and another conversion is performed at the destination. By choosing the standard to be the most common format in the system, the number of conversions can be reduced.

At a high level of abstraction the heterogeneity problem and the necessary translations are well understood. At the implementation level a number of complications exist. The issues are precision loss, format incompatibilities (e.g., minus zero value in sign magnitude and 1's complement cannot be represented in 2's complement), data type incompatibilities (e.g., mapping of an upper/lower case terminal to an upper case only terminal is a loss of information), efficiency concerns, the number of locations of the translators, and what constitutes a good intermediate data format for a given incompatibility problem.

As DCS's become more integrated, one can expect that both programs and complicated forms of data might be moved to heterogeneous hosts. How will a program run on this host, given that the host has different word lengths, different machine code, and different operating system primitives? How will database relations stored as part of a CODASYL database be converted to a relational model and its associated storage scheme? Moving a data structure object requires knowledge about the semantics of the structure (e.g., that some of the fields are pointers and these have to be updated upon a move). How should this information be imparted to the translators, what are the limitations, if any, and what are the benefits and costs of having this kind of flexibility? In general, the problem of providing translation for movement of data and programs between heterogeneous hosts and networks has not been solved. The main problem is ensuring that such programs and data are interpreted correctly at the destination host. In fact, the more difficult problems in this area have been largely ignored.

The Open Systems Foundation (OSF) distributed computing environment (DCE) is attempting to address the problem of programming and managing heterogeneous distributed computer systems by establishing a set of standards for the major components of such systems. This includes standards for RPCs, distributed file servers, and distributed management.

2.5 Efficiency

Distributed computer systems are meant to be efficient in a multitude of ways. Resources (files, compilers, debuggers, and other software products) developed at one host can be shared by users on other hosts limiting duplicate efforts. Expensive hardware resources can also be shared minimizing costs. Communication facilities, such as the remote procedure call, electronic mail, and file transfer protocols, also improve efficiency by enabling better and faster transfer of information. The multiplicity of processing elements might also be exploited to improve response time and throughput of user processes. While efficiency concerns exist at every level in the system, they must also be treated as an integrated "system" level issue. For example, a good design, the proper tradeoffs between levels, and the pairing down of over-ambitious features usually improves efficiency. In this section, however, we concentrate on discussing efficiency as it relates to the execution time of processes (threads).

Once the system is operational, improving response time and throughput of user processes (threads) is largely the responsibility of scheduling and resource management algorithms [6, 30, 32, 39, 75, 111, 112, 126], and the mechanisms used to move processes and data [28, 33, 42, 127]. The scheduling algorithm is intimately related to the resource allocator because a process will not be scheduled for the CPU if it is waiting for a resource. If a DCS is to exploit the multiplicity of processors and resources in the network, it must contain more than simply "n" independent schedulers. The local schedulers must interact and cooperate, and the degree to which this occurs can vary widely. We suggest that a good scheduling algorithm for a DCS will be a heuristic that acts like an "expert system." This expert system's task is to effectively utilize the resources of the entire distributed system given a complex and dynamically changing environment. We hope to illustrate this in the following discussion.

In the remainder of this section when we refer to the scheduling algorithm, we are referring to the part of the scheduler (possibly an expert system) that is responsible for choosing the host of execution for a process. We assume that there is another part of the scheduler which assigns the local CPU to the highest priority ready process.

We divide the characteristics of a DCS which influence response time and throughput into system characteristics, and scheduling algorithm characteristics. System characteristics include: the number, type, and speed of processors, caches, and memories, the allocation of data and programs, whether data and programs can be moved, the amount and location of replicated data and programs, how data are partitioned, partitioned functionality in the form of dedicated processors, any special purpose hardware, characteristics of the communication subnet, and special problems of distribution such as no central clock and the inherent delays in the system. A good scheduling algorithm would take the system characteristics into account. Scheduling algorithm characteristics include: the type and amount of state information used, how and when that information is transmitted, how the information is used (degree and type of cooperation between distributed scheduling entities), when the algorithm is invoked, adaptability of the algorithm, and the stability of the algorithm [24, 110].

The type of state information used by scheduling algorithms includes queue lengths, CPU utilization, amount of free memory, estimated average response time, or combinations of various information, in making its scheduling decision. The type of information also refers to whether the information is local or networkwide information. For example, a scheduling algorithm on host 1 could use queue lengths of all the hosts in the network in making its decision. The amount of state information refers to the number of different types of information used by the scheduler.

Information used by a scheduler can be transmitted periodically or asynchronously. If asynchronously, it may be sent only when requested (as in bidding), it may be piggybacked on other messages between hosts, or it may be sent only when conditions change by some amount. The information may be broadcast to all hosts, sent to neighbors only, or to some specific set of hosts.

The information is used to estimate the loads on other hosts of the network in order to make an informed global scheduling decision. However, the data received are out of date and even the ordering of events might not be known [63]. It is necessary to manipulate the data in some way to obtain better estimates. Several examples are: very old data can be discarded: given that state information is timestamped, a linear estimation of the state extrapolated to the current time might be feasible; conditional probabilities on the accuracy of the state information might be calculated in parallel with the scheduler by some monitor nodes and applied to the received state information; the estimates can be some function of the age of the state information; or some form of (iterative) message interchange might be feasible.

Before a process is actually moved, the cost of moving it must be accounted for in determining the estimated benefit of the move. This cost is different if the process has not yet begun execution than if it is already in progress. In both cases, the resources required must also be considered. If a process is in execution, then environment information (e.g., the process control block) probably should be moved with the process. It is expected that in many cases the decision will be not to move the process.

Schedulers invoked too often will produce excessive overhead. If they are not invoked often enough, they will not be able to react fast enough to changing conditions. There will be undue startup delay for processes. There must be some ideal invocation schedule which is a function of the load.

In a complicated DCS environment, it can be expected that the scheduler will have to be quite adaptive [74, 110]. A scheduler might make minor adjustments in weighing the importance of various factors as the network state changes in an attempt to track a slowly changing environment. Major changes in the network state might require major adjustments in the scheduling algorithms. For example, under very light loads there does not seem to be much justification for networkwide scheduling, so the algorithm might be turned off—except the part that can recognize a change in the load. At moderate loads, the full blown scheduling algorithm might be employed. This might include individual hosts refusing all requests for information and refusing to accept any process because it is too busy. Under heavy loads on all hosts it again seems unnecessary to use networkwide scheduling. A bidding scheme might use both source and server directed bidding [112]. An overloaded hosts asks for bids and is the source of work for some other hosts in the network. Similarly, a lightly loaded host may make a reverse bid, i.e., it asks the rest of the network for some work. The two types of bidding might coexist. Schedulers could be designed in a multilevel fashion with decisions being made at different rates, e.g., local decisions and state information updates occur frequently, but more global exchange of decisions and state information might proceed at a slower rate because of the inherent cost of these global actions.

A classic efficiency question in any system is: what should be supported by the kernel, or more generally by the operating system, and what should be left to the user? The trend in DCS is to provide minimal support at the kernel level, e.g., supporting objects, primitive IPC mechanisms, and processes (threads). Then other operating system functions are supported as higher level processes. On the other hand, because of efficiency concerns some researchers advocate putting more in the kernel including communication protocols, real-time systems support, or even supporting the concept of a transaction in the kernel. This argument will never be settled conclusively since it is a function of the requirements, type of processes running, etc.

Of course, many other efficiency questions remain. These include: the efficiency of the object model, the end-to-end argument, locking granularity, performance of remote operations, improvements due to distributed control, the cost effectiveness of various reliability mechanisms, efficiently dealing with heterogeneity, hardware support for operating system functions [7], and handling the I/O bottleneck via disk arrays of various types called RAID 1 through 6 [56, 86] (redundant arrays of inexpensive disks). Efficiency is, therefore, not a separate issue but must be addressed for each issue in order to result in an efficient, reliable, and extensible DCS. A difficult question to answer is exactly what is acceptable performance given that multiple decisions are being made at all levels and that these decisions are being made in the presence of missing and inaccurate information.

2.6 Real-Time

Real-time applications such as nuclear power plants and process control are inherently distributed and have severe real-time and reliability requirements. These requirements add considerable complication to a DCS. Examples of demanding real-time systems include ESS [13], REBUS [9], and SIFT [132]. ESS is a software controlled electronic switching system developed by the Bell System for placing telephone calls. The system meets severe real-time and reliability requirements. REBUS is a fault tolerant distributed system for industrial real-time control, and SIFT is a fault tolerant flight control system. Generally, these systems are built with technology that is tailored to these applications because many of the concepts and ideas used in general purpose distributed computing are not applicable when deadlines must be guaranteed. For example, remote procedure calls, creating tasks and threads, and requesting operating system services are all done in a manner that ignores deadlines, causes processes to block at any time, and only provides reasonable average case performance. None of these things are reasonable when it is critical that deadlines be met. In fact, many misconceptions [115] exist when dealing with distributed, real-time systems. However, significant new research efforts are now being conducted to combat these misconceptions and to provide a science of real-time computing in the areas of formal verification, scheduling theory and algorithms [72, 91], communications protocols [8, 135], and operating systems [98, 116, 125]. The goal of this new research in real-time systems is to develop predictable systems even when the systems are highly complex, distributed, and operate in non-deterministic environments [117].

Other distributed real-time applications such as airline reservation and banking applications have less severe real-time constraints and are easier to build. These systems can utilize most of the general purpose distributed computing technology described in this chapter, and generally approach the problem as if real-time computing were equivalent to fast computing which is false. While this seems to be common practice, some additions are required to deal with real-time constraints, e.g., scheduling algorithms may give preference to processes with earlier deadlines, or processes holding a resource may be aborted if a process with a more urgent deadline requires the resource. Results from the more demanding real-time systems may also be applicable to these *soft* real-time systems.

3 Distributed Databases

Database systems have existed for many years providing significant benefits in high availability and reliability, reduced costs, good performance, and ease of sharing information. Most are built using what can be called a database architecture [34, 85]. See Figure 5. The architecture includes a query language for users, a data model that describes the information content of the database as seen by the users, a schema (the definition of the structure, semantics and constraints on the use of the database), a mapping that describes the physical storage structure used to implement the data model, a description of how the physical data will be accessed, and, of course, the data (database) itself. All of these components are then integrated into a collection of software which handles all accesses to the database. This software is called the database management system (DBMS). The DBMS usually supports a transaction model. In this section we discuss various transaction structures, access and concurrency control, reliability, heterogeneity, efficiency techniques, and real-time distributed databases.

3.1 Transaction Structures

A transaction is an abstraction that allows programmers to group a sequence of actions on the database into a logical execution unit [17, 58, 65, 107]. Transactions either commit or abort. If the transaction successfully completes all its work, the transaction commits. A transaction which aborts does so without completing its operations and any effect of executed actions must be undone. Transactions have four properties, known as the ACID properties: *a*tomicity, *c*onsistency, *i*solation and *d*urability. *Atomicity* means that either the entire transaction completes, or it is as if the transaction never executed. *Consistency* means that the transaction maintains the integrity constraints of the database. *Isolation* means that even if transactions execute concurrently, their results appear as if they were executed in some serial order. *Durability* means that all changes made by a committed transaction are permanent.

A distributed database is a *single* logical database, but it is physically distributed over many sites (nodes). A distributed database management system (DDBMS) controls access to the distributed data supporting the ACID properties but with the added complication imposed by the physical distribution. For example, a user may issue a transaction which updates various data that, transparent to the user, physically resides on many different

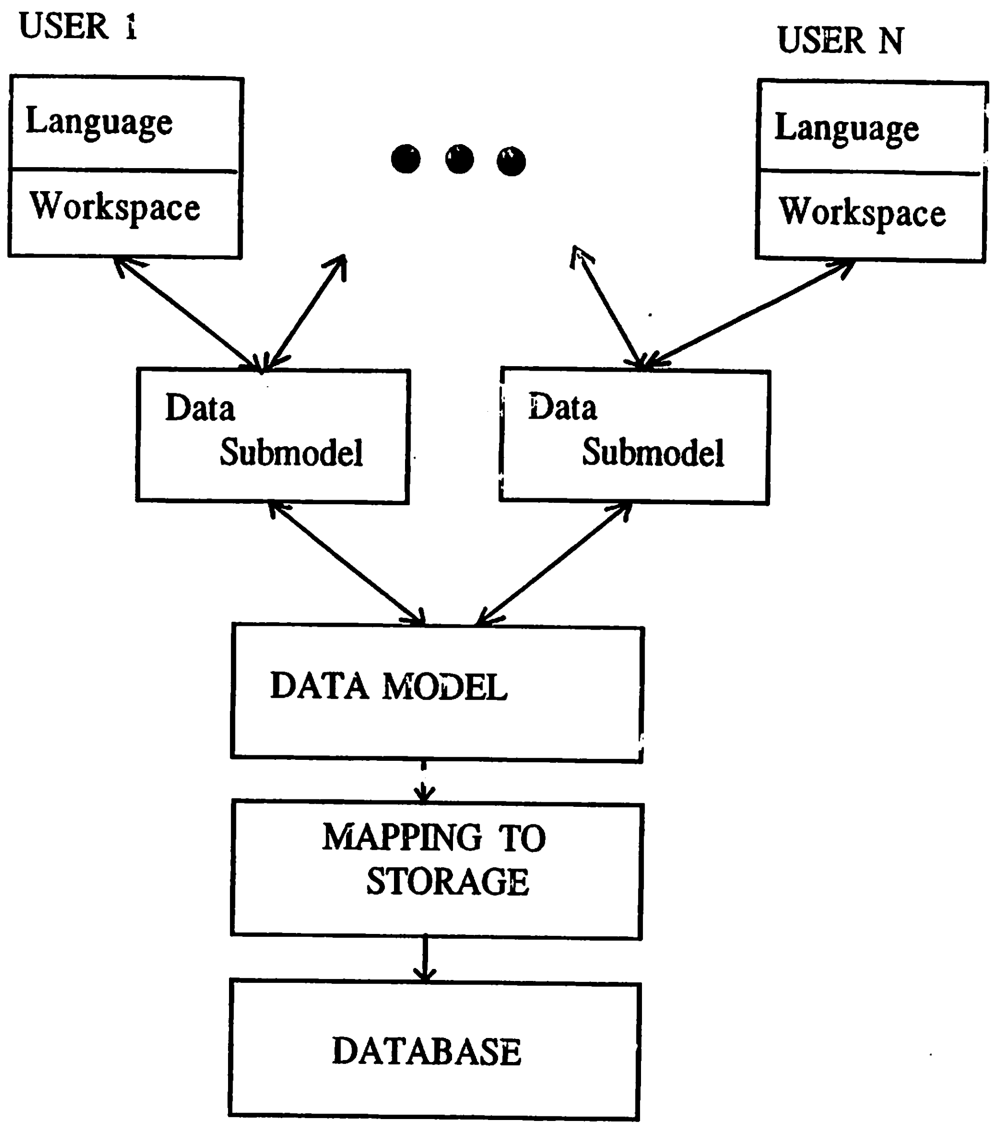


FIGURE 5: DATABASE ARCHITECTURE

nodes [118]. The software which supports the ACID properties of transactions must then interact with all these nodes in a manner that is consistent with the ACID properties. This usually requires supporting remote node slave transactions, distributed concurrency control, recovery, and commit protocols, etc. There have been many papers and books written about these basic aspects of distributed database management systems. See [15, 16, 17, 58, 85]. Rather than discussing these basic issues, we will discuss some of the extended transaction models which have been recently developed.

The traditional transaction model, while powerful because of its ability to mask the effects of failures and concurrency, has shortcomings when applied to complex applications such as computer aided design, computer aided software engineering, distributed operating systems, and multi-media databases. In these applications there is a need for greater functionality and performance than can be achieved with traditional transactions. For example, two programmers working on a joint programming project may wish for their transactions to be cooperative and to see partial results of the other user, rather than being competitive and isolated, properties exhibited by the traditional transaction model. Also, traditional transactions only exploit very simple semantics such as read-only and write-only semantics of the transactions in order to achieve greater concurrency and performance.

Many extended transaction models have been proposed to support the greater functionality and performance required by complex applications. These include nested transactions [76, 89], multi-level transactions [77, 11], cooperative transactions [60], compensating transactions [61], recoverable communicating actions [128], split transactions [90], and sagas [44, 45]. Each of these transaction models has different semantics with respect to visibility, consistency, recovery and permanence in their attempt to be useful for various complex applications. As an example, a nested transaction is composed of subtransactions that may execute concurrently. Subtransactions are serializable with respect to siblings and other non-related transactions. Subtransactions are failure atomic with respect to their parent transaction. A subtransaction can abort without necessarily causing the parent to abort. The other extended models have features such as relaxing serializability or failure atomicity, may have structures other than hierarchical, and may exhibit different abort dependencies. The relationship and utility of these models are currently being explored. In this regard, there has been the development of a comprehensive and flexible framework, called ACTA [31], used to provide a formal description and a reasoning procedure for the properties of all these extended transaction models.

There is another dimension along which traditional transactions have changed for complex applications. Initially, traditional transactions were considered as performing simple read or write operations on the database. However, there has been a merger of ideas from object based programming and database systems, resulting in object based databases [14, 22, 73]. Here, transactions and extended transactions perform higher level operations on objects which reside in the database. Using object based databases provides more support for complex applications than having to work with simple read and write operations.

3.2 Access and Concurrency Control

Most access control in distributed databases is supported by the underlying operating system. In particular, it is the operating system that verifies that a database user is who he claims to be and that controls which user is allowed to read or write various data. Users are typically grouped so that each group has the same rights. Maintaining and updating the various rights assigned to each group is non-trivial, and is exacerbated when heterogeneous databases are considered.

In a database system multiple transactions will be executing in parallel and may conflict over the use of data. Protocols for resolving data access conflicts between transactions are called concurrency control protocols [5, 15, 62, 67, 123]. The correctness of a concurrency control protocol is usually based on the concept of serializability [16]. Serializability means that the effect of a set of executed transactions (permitted to run in parallel) must be the same as *some* serial execution of that set of transactions. In many cases and at all levels the strict condition of serializability is not required. Relaxing this requirement can usually result in access techniques that are more efficient in the sense of allowing more parallelism and faster execution times. However, due to space limitations we will not discuss these extensions here. See [129, 130].

Three major classes of concurrency control protocols are: locking, timestamp ordering [95], and validation (also called the optimistic approach [62]). Locking is a well-known technique. In the database context the most common form of locking is called two-phase locking. See [16] for a description of this protocol.

Timestamp ordering is an approach where all accesses to data are timestamped and then some common rule is followed by all transactions in such a way as to ensure serializability [123]. This technique can be useful at all levels of a system, especially if timestamps are already being generated for other reasons such as to detect lost messages or failed hosts. A variation of the timestamp ordering approach is the multiversion approach. This approach is interesting because it integrates concurrency control with recovery. In this scheme, each change to the database results in a new version. Concurrency control is enforced on a version basis and old versions serve as checkpoints. Handling multiple versions efficiently is accomplished by differential files [99].

Validation is a technique which permits unrestricted access to data items (resulting in no blocking and hence fast access to data), but then checks for potential conflicts at the commit point. The commit point is the time at which a transaction is sure that it will complete. This approach is useful when few conflicts are expected because access is very fast and if most transactions are validated (due to the few conflicts), then there is also little overhead due to aborting any non-validated transactions. Most validation protocols assume a particular recovery scheme and can also be considered to integrate concurrency control and recovery.

3.3 Reliability

Recovery management in distributed databases is a complex process [58, 102, 103]. Recovery is initiated due to problems such as invalid inputs, integrity violations, deadlocks, and node and media failures. All of these faults, except node failures, are addressed by simple transaction rollback. Node failures require much more complicated solutions. All solutions are based on data redundancy and make use of stable storage where past information concerning the database has been saved and can survive failures.

It is the recovery manager component of database systems that is responsible for recovery. Generally, the recovery manager must operate under six scenarios.

1. Under normal operation the recovery manager logs each transaction and its work, and at transaction commit time checks transaction consistency, records the commit operation on the log, and forces the log to stable storage.

2. If a transaction must be rolled back, the recovery manager performs the rollback to a specific checkpoint, or completely aborts the transaction (essentially a rollback to the beginning of the transaction).

3. If any database resource manager crashes, the recovery manager must obtain the proper log records and restore the resource manager to the most recent committed state.

4. After a node crash, the recovery manager must restore the state of all the resource managers at that node and resolve any outstanding distributed transactions that were using that node at the time of the crash and were not able to be resolved due to the crash.

5. The recovery manager is responsible for handling media recovery (such as a disk crash) by using an update log and archive copies or copies from other nodes if replicated data is supported by the system.

6. It is typical that distributed databases have recovery managers at each node which cooperate to handle many of the previous scenarios. These recovery managers themselves may fail. Restart requires reintegrating a recovery manager into the set of active recovery managers in the distributed system.

Actually supporting all of the above scenarios is difficult and requires sophisticated strategies for what information is contained in a log, how and when to write the log, how to maintain the log over a long time period, how to undo transaction operations, how to redo transaction operations, how to utilize the archives, how and when to checkpoint, and how to interact with concurrency control and commit processing.

Many performance tradeoffs arise when implementing recovery management, e.g., how often and how to take checkpoints. If frequent checkpoints are taken, then recovery is faster and less work is lost. However, taking checkpoints too often significantly slows the "normal" operation of the system. Another question arises in how to efficiently log the information needed for recovery. For example, many systems perform a lazy commit where at commit time all the log records are created but only pushed to disk at a later time. This reduces the guarantees that the system can provide about the updates, but it improves performance.

3.4 Heterogeneity

As defined above, a distributed database is a single logical database, physically residing on multiple nodes. For a distributed database system there is a single query language, data model, schema and transaction management strategy. This is in contrast to a federated database system which is a collection of autonomous database systems integrated for cooperation. The autonomous systems may have identical query languages, data models, schemas and transaction management strategies, or they may be heterogeneous along any or all of these dimensions. The degree of integration varies from a single unified system constructed with new models for query languages, data models, etc., built on top of the autonomous components, to those systems with minimal interaction and without any unified models. These latter systems are called multidatabase systems. Federated systems arise when databases are developed independently, then need to be highly integrated. Multidatabase systems arise when individual database management systems wish to retain a great degree of autonomy and only provide minimal interaction.

Heterogeneity in database systems also arises from differences in underlying operating systems or hardware, from differences in DBMSs, and from semantics of the data itself. Several examples of these forms of heterogeneity follow.

Operating systems on different machines may support different file systems, naming conventions and IPC. Hardware instruction sets, data formats, and processing components may also be different at various nodes. Each of these may give rise to differences that must be resolved for proper integration of the databases.

If DBMSs have different query languages, data models, or transaction strategies, then problems arise. For example, differences in query languages may mean that some requests become illegal when issued on data in the "other" database. Differences in data models arise from many sources, including what structures each supports, e.g., relations versus record types, and from how constraints are specified. One transaction strategy might employ two-phase locking with after image logging, while another uses some form of optimistic concurrency control.

Other problems arise from semantics of the data. Semantic heterogeneity is not well understood and arises from differences in the meaning, definition, or interpretation of the related data in the various databases. For example, course grades may be defined on an [A,B,C,D,F] scale in one database and on a numerical scale in another database. How do we resolve the differences when a query fetches grades from *both* databases?

Solutions to the heterogeneity problem are usually difficult and costly both in development costs in dollars and in run-time execution costs. Differences in query languages are usually solved by mapping commands in one language to an equivalent set of commands in the other, and vice versa. If more than two query languages are involved, then an intermediate language is defined and mappings occur to and from this intermediate language. Mappings must be defined for the data model and schema integration. For example, an integrated schema would contain the description of the data described by all component schemas and the mappings between them. It is also possible to restrict what data in a given database can be seen by the federation of databases. This is sometimes called an export schema.

Solutions must also be developed for query decomposition and optimization, and global transaction management in federated database systems. The complexities found here are beyond this article. Interested readers should see [71, 101, 124].

3.5 Efficiency

Distributed database systems make use of many techniques for improving efficiency including: distributed and local query optimization, various forms of buffering, lazy evaluation, disk space allocation tailored to access requirements, the use of mapping caches, parallelism in subtransactions, and non-serializability. Parallelism in subtransactions and non-serializability have been mentioned before so these issues will not be discussed in this subsection. It is often necessary for operating systems that support databases to be specifically tailored for database functions, and if not, then the support provided by the operating system is usually inefficient. See [119] for a discussion on this issue.

To obtain good performance in distributed databases, query optimization is critical. Query optimization can be categorized as either heuristic where ad hoc rules transform the original query into an equivalent query that can be processed more quickly, or systematic where the estimated cost of various alternatives are computed using analytical cost models that reflect the details of the physical distributed database. In relational databases, JOIN, SELECTION, and PROJECTION are the most time consuming and frequent operations in queries and hence most query optimizers deal with these operations. For example, a JOIN is one of the most time consuming relational operators because the resultant size can be equal to the product of the sizes of the original relations. The most common optimization is to first perform PROJECTIONS and SELECTIONS to minimize intermediate relations to be JOINED. In a distributed setting, minimizing the intermediate relations can also have the effect of minimizing the amount of data that needs to be transferred across the network. Specialized query optimization techniques also exist for statistical databases [84] and memory resident databases [133].

Supporting a database buffer (sometimes called a database cache) is one of the most important efficiency techniques required [40, 96]. The database buffer manager acts to make pages accessible in main memory (reading from the disk) and to coordinate writes to the disk. In doing the reading and writing, it attempts to minimize I/O and to do as much I/O in a lazy fashion as possible. One example of the lazy I/O would be that at transaction commit time only the log records are forced to the disk, the commit completes and the actual data records are written at a later time when the disk is idle or if forced for other reasons such as a need for more buffer space. Part of the buffer managers's task is to interact with one log manager by writing to the log and to cooperate with the recovery manager. If done poorly, disk I/O's for logging can become a bottleneck.

Disk space allocation is another important consideration in attaining good database performance. In general, the space allocation strategy should be such that fast address translation from logical block numbers to physical disk addresses can occur without any I/O, and the space allocation should be done to support both direct and sequential access.

The full mapping of relations through multiple intermediate levels of abstractions (e.g., from relations, to segments, to OS files, to logical disks, to extents, to disk blocks) down to the physical layer must be done efficiently. In fact, one should try to eliminate unnecessary intermediate layers (still retaining data independence), and use various forms of mapping caches to speed up the translations.

3.6 Real-Time

Real-time transaction systems are becoming increasingly important in a wide range of applications. One example of a real-time transaction system is a computer integrated manufacturing system where the system keeps track of the state of physical machines, manages various processes in the production line, and collects statistical data from manufacturing operations. Transactions executing on the database may have deadlines in order to reflect, in a timely manner, the state of manufacturing operations or to respond to the control messages from operators. For instance, the information describing the current state of an object may need to be updated before a team of robots can work on the object. The update transaction is considered successful only if the data (the information) is changed consistently (in the view of all the robots) and the update operation is done within the specified time period so that all the robots can begin working with a consistent view of the situation. Other applications of real-time database systems can be found in program trading in the stock market, radar tracking systems, command and control systems, and air traffic control systems.

Real-time transaction processing is complex because it requires an integrated set of protocols that must not only satisfy database consistency requirements but also operate under timing constraints [36, 37, 50, 69]. The algorithms and protocols that must be integrated include: cpu scheduling, concurrency control, conflict resolution, transaction restart, transaction wakeup, deadlock, buffer management, and disk I/O scheduling [21, 23, 25, 26, 50, 100]. Each of these algorithms or protocols should directly address the real-time constraints.

To date, work on real-time databases has investigated a centralized, secondary storage real-time database [1, 2, 3]. As is usually required in traditional database systems, work so far has required that all the real-time transaction operations maintain data consistency as defined by serializability. Serializability may be relaxed in some real-time database systems, depending on the application environment and data properties [105, 114, 69], but little actual work has been done in this area. Serializability is enforced by using a real-time version of either the two-phase locking protocol or optimistic concurrency control. Optimistic concurrency control has been shown to perform better than two-phase locking when integrated with priority-driven CPU scheduling in real-time database systems [48, 49, 53].

In addition to timing constraints, in many real-time database applications, each transaction imparts a value to the system, which is related to its criticalness and to when it completes execution (relative to its deadline). In general, the selection of a value function depends on the application [72]. To date, the value of a transaction has been modeled as a function of its criticalness, start time, deadline, and the current system time. Here criticalness represents the importance of transactions, while deadlines constitute the time constraints of real-time transactions. Criticalness and deadline are two characteristics of real-time transactions and they are not necessarily related. A transaction which has a short deadline does not imply that it has high criticalness. Transactions with the same criticalness may have different deadlines and transactions with the same deadline may have different criticalness values. Basically, the higher the criticalness of a transaction, the larger its value to the system. It is imporatnt to note that the value of a transaction is time-variant. A transaction which has missed its deadline will not be as valuable to the system as it would be if it had completed before its deadline.

Other important issues and results for real-time distributed databases include:

- In a real-time system, I/O scheduling is an important issue with respect to the system performance. In order to minimize transaction loss probability, a good disk scheduling algorithm should take into account not only the *time constraint* of a transaction, but also the *disk service time* [25].

- Used for I/O, the *earliest deadline* discipline ignores the characteristics of disk service time, and, therefore, does not perform well except when the I/O load is low.

- Various conflict resolution protocols which directly address deadlines and criticalness can have a important impact on performance over protocols that ignore such information.

- How can priority inversion (this refers to the situation where a high priority transaction is blocked due to a low priority transaction holding a lock on a data item) be solved [52, 106]?
- How can soft real-time transaction systems be interfaced to hard real-time components?
- How can real-time transactions themselves be guaranteed to meet hard deadlines?
- How will real-time buffering algorithms impact real-time optimistic concurrency control [51]?
- How will semantics-based concurrency control techniques impact real-time performance?
- How will the algorithms and performance results be impacted when extended to a distributed real-time system?
- How can correctness criteria other than serializability be exploited in real-time transaction systems?

4 Summary

Distributed computer systems began in the early 1970s with a few experimental systems. Since that time tremendous progress has been made in many disciplines that support distributed computing. The progress has been so remarkable that DCSs are commonplace and quite large. For example, the Internet has over 500,000 nodes on it. This chapter has discussed two of the areas that played a major role in this achievement: distributed operating systems and distributed databases. For more information on distributed computing see the following good books and surveys [4, 35, 41, 46, 58, 85, 113, 115, 121]. As mentioned in the Introduction many areas of distributed computing could not be covered in this chapter. One important area omitted is distributed file servers such as NFS and Andrew. For more information on these and other distributed file servers see the survey article [68].

5 Acknowledgments

I enthusiastically thank Panos Chrysanthis and Krithi Ramamritham for their valuable comments on this work.

6 Glossary

Access Control List. A model of protection where rights are maintained as a list associated with each object.. See Capability List and Access Control Matrix.

Access Control Matrix. A model of protection where rows of the matrix represent domains of execution and columns represent the objects in the system. The entries in the matrix indicate the allowable operations each domain of execution can perform on each object.

Active Object Model. An object which has one or more execution activities (e.g., threads) associated with it at all times. Operation invocations on the object use these resident threads for execution.

Asynchronous Send. An interprocess communication primitive where the sending process does not wait for a reply before continuing to execute. See Synchronous Send.

Atomic Broadcast. A communication primitive that supports the result that either all hosts (or processes) receive the message, or none of them see the message. See Broadcasting.

Atomicity. A property of a transaction where either the entire transaction completes, or it is as if the transaction never executed. See Transaction.

Bidding. A distributed scheduling scheme that requests hosts to provide information in the form of a bid as to how well that host can accept new work.

Broadcasting. Sending a message to all hosts or processes in the system. See Multicasting.

Capability List. A model of protection where rights are maintained as a list associated with each execution domain. See Access Control List and Access Control Matrix.

Client-Server Model. A software architecture that includes server processes that provide services and client processes that request services via well-defined interfaces. A particular process can be both a server and a client process.

Consistency. A property of a transaction which means that the transaction maintains the integrity of the database. See Transaction.

Copy-on-write. Data is delayed from being copied between address spaces until the destination actually requires it.

Data Abstraction. A collection of information (data) and a set of operations on that information.

Differential Files. A representation of a collection of data as the difference from some point of reference. Used as a technique for storing large and volatile files.

Distributed Computing Environment (DCE). A computing environment that exploits the potential of computer networks without the need to understand the underlying complexity. This environment is to meet the needs of end users, system administrators, and application developers.

Distributed Operating System. A native operating system that runs on and controls a network of computers.

Distributed Shared Memory. The abstraction of shared memory in a physically non-shared distributed system.

Durability. A property of a transaction which means that all changes made by a committed transaction are permanent. See Transaction.

Federated Database System. A collection of autonomous database systems integrated for purposes of cooperation.

Hard Real-Time. Tasks have deadlines or other timing constraints and serious consequences could occur if a task misses a deadline. See Soft Real-Time.

Isolation. A property of a transactions which means that even if transactions execute concurrently, their results appear as if they were executed in some serial order. See Transaction and Serializability.

Lazy Evaluation. A performance improvement technique which postpones taking an action or even a part of an action until the results of that action (subaction) are actually required.

Lightweight Process. An efficiency technique that separates the address space and rights from the execution activity. Most useful for parallel programs and multiprocessors.

Lightweight Remote Procedure Call. An efficiency technique to reduce the execution time cost of Remote Procedure Calls when the processes happen to reside on the same host.

Multicasting. Sending a message to all members of a defined group. See Broadcasting.

Nested Transaction. A transaction model that permits a transaction to be composed of subtransactions which can fail without necessarily aborting the parent transaction.

Network Operating System. A layer of software added to local operating systems to enable a distributed collection of computers to cooperate.

Network Partitioning. A failure situation where the communication network(s) connecting the hosts of a distributed system have failed in such a manner that two or more independent subnets are executing without being able to communicate with each other.

Object. An instantiation of a data abstraction. See Data Abstraction.

Passive Object. An object which has no execution activity assigned to it. The execution activity gets mapped into the object upon invocation of operations of the object.

Process. A program in execution including the address space, the current state of the computation, and various rights to which this program is entitled.

RAID. Redundant arrays of inexpensive disks to enhance I/O throughput and fault tolerance.

Real-Time Applications. Applications where tasks have specific deadlines or other timing constraints such as periodic requirements.

Recoverable Communicating Actions. A complex transaction model to support long and cooperative non-hierarchical computations involving communicating processes.

Remote Procedure Call. A synchronous communication method which provides the same semantics as a procedure call, but it occurs across hosts in a distributed system.

Sagas. A complex transaction model for long lived activities consisting of a set of component transactions which can commit as soon as they complete. If the saga aborts, committed components are compensated.

Serializability. A correctness criterion which states that the effect of a set of executed transactions must be the same as *some* serial execution of that set of transactions.

Soft Real-Time. In a soft real-time system tasks have deadlines or other timing constraints, but no serious complications occur if a deadline is missed. See Hard Real-Time.

Split Transactions. A complex transaction model where the splitting transaction delegates the responsibility for aborting or committing changes made to a subset of objects it has accessed to the split transaction.

Synchronous Send. An interprocess communication primitive where the sending process waits for a reply before proceeding. See Asynchronous Send.

Timestamp Ordering. A concurrency control technique where all accesses to data are timestamped and then some common rule is followed to ensure serializability. See Serializability.

Thread. Represents the execution activity of a process. Multiple threads can exist in one process.

Transaction. An abstraction that groups a sequence of actions on a database into a logical execution unit. Traditional transactions have four properties. See Atomicity, Consistency, Isolation and Durability.

Validation. A concurrency control technique which permits unrestricted access to data items, but then checks for potential conflicts at the transaction commit point.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[2] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, 1988.

[3] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, 1989.

[4] B. Abeysundara and A. Kamal, "High Speed Local Area Networks and Their Performance: A Survey," *ACM Computing Surveys*, Vol. 23, No. 2, June 1991.

[5] R. Agrawal, M.J. Carey and M. Livny, "Concurrency Control Performance Modeling: Alternatives and Implications," *ACM Transaction on Database Systems*, Vol. 12, No. 4, December 1987.

[6] T. Anderson, B. Bershad, E. Lazowska, and H. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," Univ of Washington, TR, 90-04-02, October 1990.

[7] T. Anderson, H. Levy, B. Bershad, and E. Lazowska, "Interaction of Architecture and OS Design," Dept CS, TR, Univ of Washington, August 1990.

[8] K. Arvind, K. Ramamritham, and J. Stankovic, "A Local Area Network Architecture for Communication in Distributed Real-Time Systems," invited paper, *Real-Time Systems Journal*, Vol. 3, No. 2, May 1991, pp. 113-147.

[9] J. Ayache, J. Courtiat, and M. Diaz, "REBUS, A Fault Tolerant Distributed System for Industrial Control," *IEEE Transactions Computers.*, Vol. C-31, July 1982.

[10] M. Bach, N. Coguen, and M. Kaplan, "The ADAPT System: A generalized Approach Towards Data Conversion," *Proceedings VLDB*, October 1979.

[11] B. Badrinath and K. Ramamritham, "Performance Evaluation of Semantics-Based Multi-level Concurrency Control Protocols," *Proceedings ACM SIGMOD*, May 1990, pp 163-172.

[12] J. Ball, J. Feldman, J. Low, R. Rashid, and P. Rovner, "RIG, Rochester's Intelligent Gateway: System Overview," *IEEE Transactions Software Engineering*, Vol. SE-2, No. 4, December 1980.

[13] D. Barclay, E. Byrne, F. Ng, "A Real-Time Database Management System for No. 5 ESS," *Bell System Technical Journal*, Vol. 61, No. 9, November 1982.

[14] D. Batory, " GENESIS: A Project to Develop an Extensible Database Management System," *Proceedings of the International Workshop on Object-Oriented Database Systems*, September 1986, pp. 207-208.

[15] P.A. Bernstein, D.W. Shipman, and J.B. Rothnie, Jr., "Concurrency Control in a System for Distributed Databases (SDD-1)," *ACM Transactions on Database Systems*, Vol. 5, No. 1, March 1980, pp. 18-25.

[16] P. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, Reading, Mass, 1987.

[18] B. N. Bershad, T. E. Anderson, and E. D. Lazowska, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems*, Vol. 8, No. 1, February 1990, pp. 37-55.

[19] K. Birman, "Replication and Fault-Tolerance in the ISIS System," *ACM Symposium on OS Principles*, Vol. 19, No. 5, December 1985.

[20] A. Birrell, R. Levin, R. Needham, and M. Schroeder, "Grapevine: An Exercise in Distributed Computing," *Communications ACM* Vol. 25, Apr. 1982, pp. 260-274.

[21] A. Buchmann, et. al., "Time-Critical Database Scheduling: A Framework For Integrating Real-Time Scheduling and Concurrency Control," *Data Engineering Conference*, February 1989.

[22] M. Carey, et al., "The Architecture of the EXODUS Extensible DBMS," *Readings in Database Systems*, Morgan Kaufmann, 1988, pp. 488-501.

[23] M. J. Carey, R. Jauhari and M. Livny, "Priority in DBMS Resource Scheduling," *Proceedings of the 15th VLDB Conference*, 1989.

[24] T. Casavant and J. Kuhl, "A Taxonomy of Scheduling in General Purpose Distributed Computing Systems," *Transactions on Software Engineering*, Vol. 14, No. 2, February 1988.

[25] S. Chen, J. Stankovic, J. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems," *Real-Time Systems*, Vol. 3, No. 3, September 1991.

[26] S. Chen, and D. Towsley, "Performance of a Mirrored Disk in a Real-Time Transaction System," *Proceedings 1991 ACM SIGMETRICS*, May 1991.

[27] D. Cheriton, H. Goosen and P. Boyle, "Paradigm: A Highly Scalable Shared Memory Multicomputer Architecture," *IEEE Computer*, February 1991.

[28] D. Cheriton and W. Zwaenepoel, "Distributed Process Groups in the V Kernel," *ACM Transactions on Computer Systems*, Vol. 3, No. 2, May 1985.

[29] R. Chin and S. Chanson, "Distributed Object Based Programming Systems," *ACM Computing Surveys*, Vol. 23, No. 1, March 1991.

[30] T. Chou and J. Abraham, "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering*, Vol. SE-8, No. 4, July 1982.

[31] P. Chrysanthis, and K. Ramamritham, "ACTA; A Framework for Specifying and Reasoning about Transaction Structure and Behavior," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1990, pp. 194-203.

[32] W. Chu, L. Holoway, M. Lan, and K. Efe, "Task Allocation in Distributed Data Processing," *IEEE Computer*, Vol. 13, November 1980, pp. 57-69.

[33] P. Dasgupta, R. LeBlanc, M. Ahamad, and U. Ramachandran, "The Clouds Distributed Operating System," *IEEE Computer*, Vol. 24, No. 11, November 1991, pp. 34-44.

[34] C. J. Date, *An Introduction to Database Systems*, Addison Wesley, Reading, Mass., 1975.

[35] D.W. Davies, E. Holler, E.D. Jensen, S.R. Kimbleton, B.W. Lampson, G. LeLann, K.J. Thurber, and R.W. Watson, *Distributed Systems—Architecture and Implementation*, Vol. 105, Lecture Notes in Computer Science. Berlin: Springer-Verlag, 1981.

[36] U. Dayal, et. al., "The HiPAC Project: Combining Active Database and Timing Constraints," *ACM SIGMOD Record*, March 1988.

[37] U. Dayal, "Active Database Management Systems," *Proceedings of the 3rd International Conference on Data and Knowledge Management*, June 1988.

[38] J. Dion, "The Cambridge File Server," *ACM Opererating Systems Review*, October 1980.

[39] K. Efe, "Heuristic Models of Task Assignment Scheduling in Distributed Systems," *IEEE Computer*, Vol. 15, June 1982.

[40] W. Effelsberg and T. Haerder, "Principles of Database Buffer Management," *ACM Transactions on Database Systems*, Vol. 9, No. 4, December 1984.

[41] P. Enslow, "What is a Distributed Data Processing System," *IEEE Computer*, Vol. 11, January 1978.

[42] E. Felten and J. Zahorjan, "Issues in the Implementation of a Remote Memory Paging System," Univ of Washington, TR, 91-03-09, March 1991.

[43] R. Fitzgerald and R. Rashid, "Integration of Virtual Memory Management and Interprocess Communication in Accent, *ACM Transactions on Computer Systems*, Vol. 4, No. 2, May 1986.

[44] H. Garcia-Molina, D. Gawlick, J. Klein, K. Kleissner, and K. Salem, "Modeling Long-Running Activities as Nested Sagas," *IEEE Technical Committee on Data Engineering*, 14(1):14-18, March 1991.

[45] H. Garcia-Molina, and K. Salem, "SAGAS," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1987, pp. 249-259.

[46] A. Goscinski, *Distributed Operating Systems: The Logical Design*, Addison Wesley, Sdyney, 1991.

[47] J. Gray, "Notes on Database Operating Systems," *Operating Systems: An Advanced Course*, Springer-Verlag, Berlin, 1979.

[48] J. R. Haritsa, M.J. Carey and M. Livny, "On Being Optimistic About Real-Time Constraints," Principles of Distributed Computing, 1990.

[49] J. R. Haritsa, M.J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the 11th Real-Time Systems Symposium*, December 1990.

[50] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings Real-Time System Symposium*, December 1989.

[51] J. Huang and J. Stankovic, "Real-Time Buffer Management," COINS TR 90-65, Univ. of Massachusetts, August 1990.

[52] J. Huang, J. Stankovic, D. Towsley, and K. Ramamritham, "Priority Inheritance Under Two-Phase Locking," *Proceedings of the Real-Time Systems Symposium*, December 1991.

[53] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes," *Proceedings VLDB*, September 1991.

[54] N. Hutchinson and L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, Vol. 17, No. 1, January 1991.

[55] T. Joseph and K. Birman, "Reliable Broadcast Protocols," Cornell Univ, TR 88-918, June 1988.

[56] R. Katz, G. Gibson and D. Patterson, "Disk System Architectures for High Performance Computing," *Proceedings of the IEEE*, Vol. 77, No. 12, December 1989.

[57] J. P. Kearns and S. DeFazio, "Diversity in Database Reference Behavior," *Performance Evaluation Review*, Vol. 17, No. 1, May 1989.

[58] W. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys*, Vol. 13, No. 2, June 1981.

[59] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed Fault Tolerant Real-Time Systems: the Mars Approach," *IEEE Micro*, Vol. 9, No. 1, February 1989, pp. 25-40.

[60] H. Korth, W. Kim, and F. Bancilhon, "On Long-Duration CAD Transactions," *Information Sciences*, 46(1-2):73-107, October-November 1988.

[61] H. Korth, E. Levy, and A. Silberschatz, "Compensating Transactions: A New Recovery Paradigm," *Proceedings of the 16th VLDB Conference*, August 1990, pp. 95-106.

[62] H. T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, June 1981.

[63] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications ACM*, July 1978.

[64] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions Programming Language and Systems*, Vol. 4, No. 3, July 1982.

[65] B. Lampson, "Atomic Transactions," *Lecture Notes in Computer Science*, Vol. 105, Springer-Verlag, 1980, pp. 365-370.

[66] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal, "The Architecture for the Eden System," *Proceedings 8th Annual Symposium on Operating System Principles*, December 1981.

[67] G. LeLann, "Algorithms for Distributed Data-Sharing Systems Which Use Tickets," *Proceedings 3rd Berkeley Workshop on Distributed Databases and Computer Networks*, 1978.

[68] E. Levy and A. Silberschatz, "Distributed File Systems: Concepts and Examples," *ACM Computing Surveys*, Vol. 22, No. 4, December 1990.

[69] K. J. Lin, "Consistency Issues in Real-Time Database Systems," *Proceedings of the 22nd Hawaii International Conference on System Sciences*, January 1989.

[70] B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Systems," *Proceedings 9th Symposium on Principles of Programming Languages*, January 1982, pp. 7-19.

[71] W. Litwin, L. Mark, and N. Roussopoulos, "Interoperability of Multiple Autonomous Databases," *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[72] C. D. Locke, "Best-Effort Decision Making for Real-Time Scheduling," *Ph.D. Dissertation*, Canegie-Mellon University, 1986.

[73] D. Maier, J. Stein, A. Ottis, and A. Purdy, "Development of an Object-Oriented DBMS," *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, October 1986, pp. 472-482.

[74] R. Mirchandaney, D. Towsley, and J. Stankovic, "Adaptive Load Sharing in Heterogeneous Distributed Systems," *Journal of Parallel and Distributed Computing*, Vol. 9, September 1990, pp. 331-346.

[75] R. Mirchandaney, D. Towsley, and J. Stankovic, "Analysis of the Effects of Delays on Load Sharing," *IEEE Transactions on Computers*, Vol. 38, No. 11, November 1989, pp. 1513-1525.

[76] J. E. B. Moss, *Nested Transactions: An Approach to Reliable Distributed Computing*, PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[77] J. E. B. Moss, N. Griffeth, and M. Graham, "Abstraction in Recovery Management," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, May 1986, pp. 72-83.

[78] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren, "Experiences with the Amoeba Distributed Operating System," *Communications of the ACM*, Vol. 33, No. 12, December 1990.

[79] R.M. Needham and A.J. Herbert, *The Cambridge Distributed Computing System*, London: Addison-Wesley, 1982.

[80] R. M. Needham and M. Schroeder, "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, Vol. 21, No. 12, December 1978, pp. 993-999.

[81] B.J. Nelson, "Remove Procedure Call," Xerox Corp., Tech. Rep. CSL-81-9, May 1981.

[82] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer*, Vol. 24, No. 8, August 1991, pp. 52-60.

[83] J. Ousterhout, D. Scelza, and P. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Communications ACM*, Vol. 23, February 1980.

[84] G. Ozsoyoglu, V. Matos, Z. Meral Ozsoyoglu, "Query Processing Techniques in the Summary-Table-by-Example Database Query Language," *ACM Transactions on Database Systems*, Vol. 14, No. 4, 1989, pp. 526-573.

[85] M. Ozsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice Hall, Englewood Cliffs, N.J., 1991.

[86] D. Patterson, G. Gibson and R. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proc ACM SIGMOD*, July 1988.

[87] G. Popek *et al.*, "LOCUS, A Network Transparent, High Reliability Distributed System," *Proceedings 8th Symposium Operating System Principles*, December 1981, pp. 14-16.

[88] D. Powell, et. al. "The Delta-4 Distributed Fault Tolerant Architecture," Laboratoire d'Automatique et d'Analyse des Systemes, Report No. 91055, February 1991.

[89] C. Pu, *Replication and Nested Transactions in the Eden Distributed System*, PhD thesis, University of Washington, 1986.

[90] C. Pu, G. Kaiser, and N. Hutchinson, "Split-Transactions for Open-Ended Activities, *Proceedings of the 14th International Conference on VLDB*, September 1988, pp. 26-37.

[91] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms For Real-Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Computing*, Vol. 1, No. 2, April 1990, pp. 184-194.

[92] R.F. Rashid and G.G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings 8th Symposium Operating System Principles*, December 1981.

[93] R. Rashid, "Threads of a New System, UNIX Review," August 1986, pp 37-49.

[94] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew, "Machine Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures," *IEEE Transactions on Computers*, Vol. 37, No. 8, August 1988.

[95] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, "System Level Concurrency Control. for Distributed Database Systems," *ACM Transactions on Database Systems*, Vol. 3, No. 2, June 1978.

[96] G. M. Sacco and M. Schkolnick, "Buffer Management in Relational Database Systems," *ACM Transaction on Database Systems*, Vol. 11, No. 4, December 1986.

[97] J.H. Saltzer, D.P. Reed, and D.D. Clark, "End-to-end Arguments in System Design," *Proceedings 2nd International Conference Distributed Computing Systems*, April 1981.

[98] K. Schwan, A. Geith, and H. Zhou, "From Chaos(Base) to Chaos(Arc): A Family of Real-Time Kernels," *Proceedings of the Real-Time Systems Symposium*, 1990, pp. 82-91.

[99] D.G. Severance and G.M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Transactions on Database Systems*, Vol. 1, No. 3, September 1976.

[100] L. Sha, R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[101] A. Sheth and J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[102] D. Skeen, "Nonblocking Commit Protocols," *Proceedings ACM SIGMOD*, 1981.

[103] D. Skeen and M. Stonebraker, "A Formal Model of Crash Recovery in a Distributed System," *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 3, May 1983.

[104] D. Skeen, "A Decentralized Termination Protocol," *Proceedings 1st IEEE Symposium on Reliability in Distributed Software Database Systems*, 1981.

[105] S. H. Son, "Using Replication for High Performance Database Support in Distributed Real-Time Systems," *Proceedings of the 8th Real-Time Systems Symposium*, December 1987.

[106] S. H. Son and C.H. Chang, "Priority-Based Scheduling in Real-Time Database Systems," *Proceedings of the 15th VLDB Conference*, 1989.

[107] A.Z. Spector and P.M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing," *ACM Operating System Review*, Vol. 17, No. 2, April 1983.

[108] S.K. Shrivastava, "On the Treatment of Orphans in a Distributed System," *Proceedings 3rd Symposium Reliability Distributed Systems*, October 1983.

[109] S. K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism," *Transactions on Computers*, Vol. C-31, July 1982.

[110] J. A. Stankovic, "Simulations of Three Adaptive Decentralized Controlled, Job Scheduling Algorithms," *Computer Networks*, Vol. 8, No. 3, June 1984, pp. 199-217.

[111] J. A. Stankovic, Bayesian Decision Theory and Its Application to Decentralized Control of Job Scheduling," *IEEE Transactions on Computers*, Vol. C-34, January 1985.

[112] J.A. Stankovic and I.S. Sidhu, "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups," *Proceedings 4th International Conference Distributed Computing*, May 1984.

[113] J.A. Stankovic, "A Perspective on Distributed Computer Systems," *Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp. 1102-1115.

[114] J. A. Stankovic and W. Zhao, "On Real-Time Transactions," *ACM SIGMOD Record*, March 1988.

[115] J. Stankovic, "Misconceptions About Real–Time Computing: A Serious Problem For Next Generation Systems," *IEEE Computer*, Vol. 21, No. 10, October 1988, pp. 10-19.

[116] J. Stankovic and K. Ramamritham, "The Spring Kernel: A New Paradigm for Real-Time Systems," *IEEE Software*, Vol. 8, No. 3, May 1991, pp. 62-72.

[117] J. Stankovic and K. Ramamritham, "What is Predictability for Real-Time Systems - An Editorial," *Real-Time Systems Journal*, Vol. 2, December 1990, pp. 247-254.

[118] M. Stonebraker and E. Neuhold, "A Distributed Database Version of INGRES," *Proceedings Berkeley Workshop on Distributed Data Management and Computer Networks*, 1977, pp. 19-36.

[119] M. Stonebraker, "Operating System Support for Database Management," *Communications of the ACM*, Vol. 24, July 1981, pp. 412-418.

[120] H. Sturgess, J. Mitchell, and J. Isreal, "Issues in the Design and Use of Distributed File System," *ACM Operating System Review*, July 1980.

[121] A. Tanenbaum and R. van Renesse, "Distributed Operating Systems," *ACM Computing Surveys*, Vol. 17, No. 4, December 1985, pp. 419-470.

[122] M. Theimer, K. Lantz, and D. R. Cheriton, "Preemptable Remote Execution Facility for the V-System," *Proceedings of the 10th Symposium on OS Principles*, Orcas Island, WA, December 1985, pp. 2-12.

[123] R. H. Thomas, "A Majority Consensus Approach on Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems.*, Vol. 4, No. 2, June 1979, pp. 180-209.

[124] G. Thomas, et. al., "Heterogeneous Distributed Database Systems for Production Use," *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[125] H. Tokuda and C. Mercer, "Arts: A Distributed Real-Time Kernel," *ACM Operating Systems Review*, July 1989, pp. 29-53.

[126] D. Towsley, G. Rommel, and J. Stankovic, "Analysis of Fork-Join Program Response Times on Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 3, July 1990, pp. 286-303.

[127] R. Vaswani and J. Zahorjan, "Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," Univ of Washington, TR, 91-03-03, March 1991.

[128] S. Vinter, K. Ramamritham, and D.Stemple, "Recoverable Actions in Gutenberg," *Proceedings of the 6th International Conference on Distributed Computing Systems*, May 1986, pp. 242-249.

[129] W. Weihl, "Specification and Implementation of Atomic Data Types," PhD Thesis, MIT, March 1984.

[130] W. Weihl, "Commutativity-Based Concurrency Control for Abstract Data Types," *Transactions on Computers*, Vol. 37, No. 12, december 1988, pp. 1488-1505.

[131] M. Weinstein, T. Page, B. Livezey, and G. Popek, "Transactions and Synchronization in a Distributed Operating System," *ACM Symposium on OS Principles*, Vol. 19, No. 5, December 1985.

[132] J. Wensley, L. Lamport, et. al., "SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, October 1978, pp. 1240-1255.

[133] K. Whang, and R. Krishnamurthy, "Query Optimization in a Memory-Resident Domain Relational Calculus Database System," *ACM Transactions on Database Systems*, Vol. 15, No. 1, March 1990, pp. 67-95.

[134] K. Wu and W. Fuchs, "Recoverable Distributed Shared Virtual Memory," *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990.

[135] W. Zhao, J. Stankovic, and K. Ramamritham, "A Window Protocol for Transmission of Time Constrained Messages," *IEEE Transactions on Computers*, Vol. 39, No. 9, September 1990, pp. 1186-1203.