M.I.T. LABORATORY FOR COMPUTER SCIENCE October 2, 1980

Computer Systems Research Request for Comments No. 200

DISTRIBUTED PROCESSING IN A NETWORK OF COOPERATING AUTONOMOUS COMPUTERS

by Liba Svobodova

The attached paper is a preprint of a paper to be presented at AICA (Associazione Italiana per il Calcolo Automatico) Bologna '80.

DISTRIBUTED PROCESSING IN A NETWORK OF COOPERATING AUTONOMOUS COMPUTERS

Liba Svobodova
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts, USA

Abstract

This paper surveys various issues that arise in design of a distributed computer system where individual nodes retain a high degree of autonomy. Such a distributed system is believed to be a natural realization for many applications, especially distributed processing needed to support office automation. The paper focuses on the operating system level, and presents the major goals for a programming language (programming system) for development of distributed applications. The paper also surveys the problem of distributed update, in the context of both the partitioned database and the replicated database, since understanding of this problem is important in deciding the amount of support that should be provided by the operating system.

This research was sponsored by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under Contract No. N00014-75-C.

1. Introduction

Distributed systems can be divided roughly into two categories:

a) Systems that look to their users and behave at their interfaces as if they were monolithic central systems, but are actually engineered from physically distributed pieces.

b) Systems that facilitate communication and cooperation among autonomous nodes. The users are aware of the distributed nature of the system although they should not have to be concerned with the physical location of data and services used by their programs.

The first category represents systems that are quite similar to loosely coupled multiprocessor systems (i.e., multiprocessor systems where processors do not share memory). The distributed organization is motivated primarily by cost, reliability, and performance. The primary distinction between a multiprocessor system and a distributed system lies in the communication medium. Multiprocessor systems use I/O buses that usually constrain the physical separation of the processors and limit the expansibility. Distributed systems use a general message oriented communication network that supports a variety of communication patterns and affords practically unlimited growth.

The main distinction between the first and the second category is how the system is administered. In the first case, the administration is centralized. In the second case, individual nodes have a substantial degree of autonomy, that is, the owner of a node has control over the structure of the node and its use while the node participates in a distributed application [SVOB 79a, CLAR 79]. Such systems are a natural realization for many applications. Specifically, the notion of autonomy is very important to the success of a computerized office, where sociological and political reasons pressure towards decentralization of control [DOLI 77]. However, it is beneficial to built distributed systems that support local autonomy even for applications that are more coherent, since a node designed to be autonomous is more robust with respect to errors and attacks from outside of itself and can operate (at least to some degree) even while it is completely cut off from the rest of the system.

This paper concentrates on the problems of designing distributed systems that preserve a fair amount of autonomy for the individual nodes. Its main focus is the operating system and related tools that are available to the builders of distributed applications. It should be noted that the applications implemented on this kind of systems may give their users an impression of a monolithic central system, that is, the end users do not have to be aware of the distribution. However, the distributedness is visible to the application builders: the interface that the system provides to these users is an important part of the overall design.

The kind of distributed systems considered here typically have three classes of components: the computing nodes, special servers, and a communication substrate. As an extreme, each computing node is dedicated to one individual (i.e., it is a "personal computer") or to a single application task.

The servers are specialized machines that perform particular services for the community as a whole. The servers are a direct outcome of the decision to afford significant autonomy to the individual computing nodes. It is not possible to assume that individual computing nodes are always willing to perform an operation requested by a computation running on another node; any such interaction may first have to be negotiated. Functions that are crucial to the correct and efficient operation of the whole system thus ought to be provided by special servers. Some servers will be part or an extention of the operating system; examples are file servers [PAXT 79, SWIN 79, REED 80], name lookup servers, and authentication servers. Other servers will be application specific, such as registration servers in an electronic mail system [LEVI 79]. The notion of a server is useful also in a context of a more closely coupled form a distributed system [WILK 80]. Finally, the communication substrate permits the exchange of messages between the various computing nodes and servers. The communication substrate may also include some special servers for message spooling and monitoring and diagnoses of the physical communication lines, but these servers are in general invisible to the application builders.

The major application area for these systems is *office automation* [ELLI 80]. A fully automated office will be made up of many highly autonomous tasks that execute concurrently and that interact with other tasks and with human users. Office automation includes several forms of distributed processing: management of distributed databases, high-level protocols for electronic mail and document management, real-time interactions of a group of people (teleconferencing), distributed decision making. Another important application area is distributed sensing and control, for example, aircraft tracking or city traffic control, or, more general, distributed problem solving [LESS 79]. These applications are narrower than office automation (the whole system is usually dedicated to a single specific problem area), but require a much higher degree of interaction among individual nodes, since the problem has to be solved in real time.

Since shared data are an important ingredient of these applications, the next section present a summary of the problems of distributed data management. Section 3 concentrates on the function and mechanisms of the operating system needed to support distributed processing among autonomous nodes. Section 4 looks at the problem of programming distributed systems.

2. Management of Distributed Data

Management of distributed data is an important research subject, since any distributed application includes some form of a distributed database, as does the operating system for a network of computers; thus the term database will be used here in a rather general sense.

A database is distributed by partitioning it into several pieces that reside on different physical nodes; these pieces can be viewed and managed as separate logical (abstract) objects. The whole database or just individual pieces can be also replicated, that is, copies of those data may exist at two or more nodes.

The basic problems concerning distributed databases are:

i. Physical distribution. This includes decisions regarding how the data should be partitioned among the nodes in the network, which parts should be replicated and how many copies of replicated data should be maintained and where.

ii. Internal management. This includes primarily access synchronization and recovery.

Although the operating system can help in finding a suitable physical distribution of data by supporting various measurement tools, this topic is out of the scope of this paper; the rest of this section concentrates on the issue of the internal management of distributed databases.

Much of the reasearch on distributed systems is concerned with the problem of performing an update that involves several physical nodes. Such an update must preserve certain consistency constraints. In case of partitioned databases, the consistency constraints are *internal*, that is, they are determined by the semantics of stored data, and have to be defined as part of the system specification. For replicated data, the problem is *mutual* consistency, that is, the individual copies must be consistent with each other. In both cases, it is unreasonable to require that the consistency constraints will hold at every instant; practical update algorithms must be allowed to produce brief inconsistencies as part of changing the whole database to a new consistent state. However, if all actions that need to be performed to step a database from one consistent state to another are grouped together in a single *atomic action*, the consistency constraints are guaranteed to hold between these atomic action. In the context of database systems, atomic actions are called usually transactions. While not all distributed applications will require such rigorous control as is implied by the protocols that have emerged from this body of work, mechanisms for performing distributed updates atomically belong among the basic mechanisms of a distributed operating system.

The definition of an atomic action is that it is indivisible; the temporal inconsistency of the database on which it operates is hidden within the action. This means that:

i. Atomic actions must be indivisible with respect to the effects observable by concurrent computations.

ii. Atomic actions must be indivisible with respect to failures; an atomic action is either carried to its completion, according to its specification or, if it fails, or if the originator decides to abort it, it leaves the system in the state it was prior to the invocation of that action.

The recovery aspect is in general nontrivial, even in the absence of concurrent computations, since the various pieces of a computation running on different autonomous nodes may lose contact with each other, or some may fail while others complete succesfully. To ensure indivisibility of distributed computations in the face of failures, it is necessary to use a two-phase commit protocol

[GRAY 78, LAMP 79, REED 78, MONT 78, TAKA 79, TRAI 79, LIND 79, STON 78, HAMM 79]. If concurrent operations on the same database are allowed, the need to undo the effects of individual operations must be considered in the design of the synchronization protocols; thus the two aspects of indivisibility cannot be completely separated.

Many different protocols for atomic actions have been proposed that differ in how they implement:

a) serializability [ESWA 76, TRAI 78] of atomic actions:
   i. synchronization (mutual exclusion) of accesses to individual data objects
   ii. detection and resolution of scheduling conflicts

b) recoverability of individual atomic actions:
   i. physical update of individual data objects
   ii. release of the data objects updated by individual atomic actions

Synchronization of concurrent operations that guarantees serializability can be performed in a variety of ways:

1) locking:
   a) centralized [MENA 78]
   b) distributed with centralized deadlock detection [STON 78]
   c) distributed with distributed deadlock detection [GRAY 78]

2) timestamp-based scheduling of accesses [TAKA 79, REED 78]

3) organization of the database that enforces correct ordering of messages [MONT 78].

Locking schemes, unless some additional sequencing information is used, are all vulnerable to deadlock. Deadlock detection in a distributed system may be an expensive proposition, unless locking is centralized (e.g., all requests for locks go through a centralized controller) in which case locking itself might be too expensive because of this extra step (additional messages) needed to acquire the locks. Timestamps define a complete ordering on all the operations in the system. All requests that belong to the same atomic operations carry the same timestamp, and the individual nodes in the system are required to process received requests in the timestamp order. Timestamp-based synchronization schemes vary in how they handle out of order (outdated) requests. One approach is to reject a delayed (older) request if a newer request (that is, a request with a higher timestamp) has already been processed and abort the atomic action that generated the delayed request [REED 78]. However, this may lead to a "dynamic deadlock" where the same set of transactions is aborted over and over because those transactions repreatedly outdate each other -- this is similar to the collision problem in contention networks such as Ethernet [METC 76]. An alternative solution, less susceptible to dynamic deadlock, is to reject a newer request in favor of a delayed older request, given that the atomic action that generated this newer request has not yet

been locally committed [TAKA 79].

A related problem is when the individual data objects used within an atomic action can be released. This problem has two subproblems: when the modifications to the data are made permanent, and when other users are allowed to see these modifications. To be able to recover an atomic operation, its update requests must be processed in such a way that it is possible to restore the previous content of all involved data objects. On the other hand. once a node agrees to perform the requested updates, it must guarantee that those updates will indeed be performed correctly if that atomic action is committed. Thus, at some point during the execution of an atomic action, both the new and the old value of all involved data objects must be remembered [LAMP 79, GRAY 78, GRAY 79]. Other computations may be allowed to view the updated objects before the updates are committed, but that necessitates recovery of all such computations should the one that produced the changes have to be recovered. This approach is used by Montgomery [MONT 78, MONT 79] and Takagi [TAKA 79]. Although additional mechanisms are necessary to keep track of the dependencies of the transactions that have read uncommitted data, this provision can significantly improve performance of the system, especially if the time between writing a new value of a data object and the actual commitment of the responsible action is long.

Replicated data present additional problems. A two-phase commit protocol could be used again to preserve the consistency constraint (mutual consistency). However, one of the main reasons for replicating data is the availability of such data objects even if some nodes are not operating or accessible; a two-phase commit protocol is unsuitable since it requires nearly simultaneous availability of all the copies in order to accomplish an update. In fact, if a straightforward two-phase commit protocol were used, the availability of a replicated data object would be lower than the availability of a non-replicated (single copy) data object! Thus it is necessary to use a more sophisticated scheme where only the fraction of the images must be simultaneously available to be able to proceed with the update [ALSB 76, THOM 79, GARC 78, GIFF 79, LIND 79]. Such protocols are further complicated by the requirement that update requests can be directed to *any* of the copies; careful synchronization is needed since concurrent requests could produce inconsistencies if updates were applied in different order at different nodes.

A natural question to ask is how to choose from the variety of schemes developed to solve the problem of distributed update. However, there is a more fundamental question, and that is, if and when distributed and replicated data should be used. In principle, the distributed atomic update is in conflict with the requirement of autonomy. Within any two-phase commit protocol, there exists a time window during which a participating node must abandon its autonomy and await the final decision. Some failures during this window will leave the participating nodes unable to either abort the update or commit it; such a situation may last for an extended period of time. Although protocols were developed that substantially reduce the probability that the individual nodes will not be able to proceed because of a single or of multiple failures, the price is a very high complexity of the update protocol; not only such protocols are likely to be expensive, but their compexity makes

them prone to errors. Thus distributed applications should be carefully analyzed to determine whether the consistency of the underlying database cannot be ensured by some other means. An example of a distributed application where consistency of the underlying database is assured by *compensation* rather than by a two-phase commit protocol can be found in [LISK 79]. As for replicated data, it is often the case that the individual copies needed to provide sufficient *availability of information* do not have to be all the most current version of the actual data. If it is necessary to ensure high availability of the current version of the data, a master/backup arrangement, where all updates must be directed to the same, master copy, can be used [ALSB 76]. A yet another approach is to abandon the requirement of consistency of the stored data and rely on catching the possible inconsistencies on a higher level. This approach has been used successfully in problem solving systems that already must be prepared to deal with miscellaneous inconsistencies arising from the imperfect or conflicting inputs to the system; additional inconsistencies in the implementation can be handdled by the same means [LESS 79].

The final question is how the operating system should support distributed and replicated data. The operating system should provide mechanisms that simplify implementation of distributed atomic actions, but should not (and cannot, since many consistency constraints are application dependent) enforce such atomicity automatically. The specific mechanisms will be discussed in the following section. Replication of data is often needed *within* the operating system, to provide sufficient reliability. In addition, the operating system can provide an abstraction of a replicated data object to application programs, but a simple master/backup scheme is probably sufficient.

3. Operating System for Distributed Processing

The fundamental problem in designing an operating system for distributed processing is how to provide coherence in communication among the nodes in the network while these nodes retain their autonomy [SALT 78a]. More specifically, it is necessary to decide on the appropriate level at which coherence ought to be enforced. A related question, that was already brought up in the preceding section, is how much support such an operating system ought to provide, that is, what kind of environment and tools ought to be offered to the implementors of distributed application systems.

In general, an operating system embodies *mechanisms* and *policies*. Much has been said in the literature about the need to separate these two components. Such a separation seems to be even more important in the case of distributed systems. The mechanisms provided in individual nodes determine what kind of cooperation is possible. Policies specify how the operations of the system are controlled. Of course, the mechanisms must be sufficiently powerful to support the desired policies.

Policies can be classified as *resource management policies* that control use of a single logical resource (such a logical resource may be represented by a set of separately distinguishable and controllable components) or a collection of resources and *communication policies* that control communication between system components and between a resource and its users. Communication policies are

generally called *protocols*. Thus in the following discussion the term policy will be used to imply resource management (resource allocation, protection, maintenance). In the extremes, the policies can be either global, that is, they must be observed by the whole system, or local where each node sets its own policies.

The choice between a local or a global policy depends on the type of resource being controlled. To support autonomy, each node of a distributed system ought to have a full control of its local resources. In particular, the assumption of local autonomy precludes load sharing at the level of hardware processor allocation: an autonomous node cannot be forced by some central authority to execute somebody else's program. Consequently, at this level the resource management policies ought to be *local*. Load sharing, if desired, can take place at a higher level, the *service level*. At this level, a distributed system can be viewed as a network of servers and clients, with possibly more than one instance of a server of a particular type. The decision which of the server instances is to be used when a client asks for a particular service can be based on their momentary availability and load. The actual control of the server allocation can be either centralized or distributed; in the latter case, selection of a particular server can be negotiated by a bidding protocol [FARB 72, SMIT 80]. A server can also provide undesignated computing power, that is, it can run arbitrary user programs and thus act as a direct extention of the user machine [STRO 79].

To be able to operate as an autonomous entity, each node has to have its own (copy of) operating system kernel that provides the mechanisms for allocating and deallocating the local hardware resources, controlling attached I/O devices, creating, controlling and destroying processes, and sending and receving packets of information via a communication network. An example of an operating system for an autonomous node, a personal computer that is to be used as a building block of a distributed system, is the Pilot [REDE 79]. The interprocess communication primitives in Pilot support client/server mode of communication, but a more general type of communication can also be achieved. The communication facilities are an integral part of the node kernel and interface smoothly with the management of the local resources.

Many functions needed for *distributed computing* can be conçentrated in special servers, but to smooth out the differences between remote and local services, appropriate interfaces should be provided in the individual nodes [REED 80]. A collection of such interfaces and the underlying protocols forms a *particle of the network operating system*. At each node, the operating system should provide a set of higher level primitives that hide the implementation differences of local and remote entities, or even the existence and use of outside servers, and facilitate uniform naming of both local and remote entities in the language(s) used by the application builders. The network operating system includes, in addition to the particles present in the individual computing nodes, facilities for managing a variety of servers, some of which may be invisible to the users.

Let us look at some specific issues in the design of an operating system for distributed computing that have been and still need to be studied:

▸ **Naming.** Any object in a system (data object, program, communication port, I/O device, etc.) can have several names, valid in the context of a specific computation, but ultimately such names must resolve to a name that is unique in the whole system [SALT 78b]. In many systems, this unique name for stored objects is their physical address. However, the physical address can change during the lifetime of an object In order to avoid the problem of dangling references, names of this kind should not be propagated beyond the boundaries of individual nodes. Interesting mechanisms that isolate references between separately managed storage areas were devised by Bishop [BISH 77], but such mechanisms, besides being expensive, are again incompatible with the autonomy requirement. Objects should be assigned identifiers that are unique in *space* (within the system) and that do not change during the lifetime of an object. For easier management, it is also desirable that such names be unique in *time*, that is, that they are never reused. To satisfy the autonomy requirement, each node should be able to decide independently how to name its local objects; system-wide unique identifiers are then obtained by appending the uid of the node to the uid of the object [ABRA 80, SOLL 79].

The problem of naming resources, and in particular, data objects, is closely connected to the issue of moving and copying objects. When an object is moved, it can either retain its name [ABRA 80], or it can be assigned a different name, a unique identifier local to the target node [SOLL 79]. An object may contain references to other objects (contain other objects by reference). If these references are globally unique names, copying or moving can be performed directly. Otherwise, proper interpretation and translation of the embedded names becomes necessary [SOLL 79]. A related question is whether it is necessary to also copy all the contained objects or if it is possible to defer this step until the particular contained object is actually needed, in which case it can be either copied or operated upon at the remote node. The design choices at this level strongly influence the low level support mechanisms that the system must provide for locating, manipulating, and protecting objects. Autonomy plays an important part here, since it is not possible to rely on later availability of the original site of the copied object to resolve the names of the contained objects.

When a *copy* of an object is made, it can be treated either as a separate object, unrelated to the original object, or some form of relationship with the original object can be maintained. Replicated database where mutual consistency must be maintained represents an extreme case of such a bond among the existing copies. However, as already discussed, for many applications it is sufficient to distribute *versions* of data objects that are known to represent a particular state of a data object (state that existed at a particular instance of time) but are not automatically updated when the master copy of the data is updated. A model where objects are implemented as *histories* of their states was developed by Reed [REED 78]. Each distinct state is represented by an immutable self-identifying version [SVOB 80]; these versions can be freely copied. A natural extention of this model is a mechanism that, given a local copy of some version, checks whether this version is still current, and if not, retrieves the *current* version [WYLE 79].

▸ **Name binding.** Name binding entails resolving a reference to an abstract object (a higher level machine-sensible name or a character-string name) by replacing it with information sufficient to access the physical representation of the object. Binding strategies employed in the system have a strong influence on how easy it is to dynamically reconfigure the system [ABRA 80]. Binding can be static, that is, a higher level name can be permanently bound to a particular physical object, or dynamic, when the desired object is determined only when the name is actually used. Once a higher level name is resolved to indicate a unique object within the system, the unique low level name of this object can be used in the following references. However, because of the local autonomy, the named object may be deleted or moved, thus invalidating the name. Individual nodes must be able to detect such invalid references; for this reason, the object identifiers should be unique not just for the object's lifetime, but forever, that is, they should not be reused. On the other hand, the requestor that submitted an illegal reference should be able to recover gracefully, possibly by finding (rebinding to) another instance of the requested entity in the network [ABRA 80].

▸ **Synchronization and recovery.** The problem of synchronization in distributed systems has been studied primarily in the context of distributed databases, and, as discussed in the preceding section, synchronization has to encompass the problem of recovery. To support execution of arbitrary computations as atomic actions, the operating system should provide mechanisms for grouping together updates on many different and distributed data objects, making objects recoverable, and committing/aborting the updates all at once. Reed proposed a very elegant comprehensive set of mechanisms to solve this problem [REED 78, REED 79]. The basis is formed by the already mentioned object model that captures the entire history of a data object. A request to update an object results in a new version being created. A version has a time attribute that specifies its range of validity, that is, the time interval in the history of the object during which the object had that particular value represented by that version. A version is only tentative until the action that created it is committed. If that computation fails, the version is simply discarded. The synchronization (mutual exclusion) of atomic actions is controlled through assignment of pseudo-temporal environments to atomic actions. All requests that belong to a specific atomic action are assigned pseudo-times from the same pseudo-temporal environment and the range of validity of each version is specified in pseudo-time. Finally, a special data structure called a commit record is used to supervise commitment or abortion of the changes made by the individual atomic actions. These mechanisms give the users substantial flexibility in how to resolve scheduling conflicts and how to respond to the unavailability of some resources. A reliable and efficient implementation of these mechanisms is being studied [REED 80, SVOB 80].

▸ **Reliability.** A distributed (decentralized) system is often chosen on the grounds that it is (or, can be made) more reliable than a monolithic centralized system. First, both functions and data can be replicated on independent hardware. Second, propagation of low level errors is restricted by physical separation of processes and resources. However, physical distribution,

decentralized control and autonomy requirements present new problems, in particular in regards to recovery. This issue was already discussed in connection with atomic actions, but there are additional considerations.

In order to ensure termination of distributed computations, a process waiting for a response from another node must be allowed to time out if it does not receive a response for an unusually long time. Such an action, however, may leave the system in an inconsistent state, since it is not known what the requestee has accomplished. It should be possible to retransmit a message without running into the problem that some action, if performed more than once, will produce wrong results. Thus all actions should be either idempotent (repeatable) or it must be possible to detect duplicate requests. The operating system does not always know that something is a duplicate request, thus this cannot be handled automatically. However, the operating system can provide mechanisms that ease the task of detecting duplicates on the application level: examples of such mechanisms are unique identifiers for inter-node messages and a stable message log.

While each node ought to implement correctly the protocols needed for distributed computation, no node should *rely* on the proper behavior of the rest of the system; run-time checking of incoming requests should be the normal mode of operation. Again, this task can be aided by the operating system, but it is necessary to provide the application builders with good tools for error reporting [SVOB 79b].

▸ **Protection.** In the class of distributed systems considered here, intra-node protection mechanisms are not required to have power sufficient to protect against subversion and malice. This is in strong contrast to a system such as Multics [SALT 74], and many other time-shared and multiprogrammed systems that were designed to operate properly with a set of mutually hostile users. What is required within a single node is a mechanism that protects against error and forgetfulness. Inter-node protection, on the otherhand must be able to deal with the potentially hostile environment: 1) individual nodes are autonomous, that is, it is not possible to assume that they will behave as desired by other nodes, and 2) the communication lines between nodes in general cannot be physically secured. The solutions to the second problem generally use encryption [KENT 76]. A crucial part of inter-node protection is mutual authentication of the communication parties; protocols based on encryption authentication (key distribution) were developed in [NEED 78 KENT 80a]. A different kind of problem is that when complex data structures are transfered between nodes, selected fileds may have to be "covered" (e.g., nil or zero can be sent instead). Again, the operating system can perform this covering automatically as part of the data transfer/copy protocol [SOLL 79].

A new consideration in distributed systems that allow extensive autonomy and where the individual nodes cannot be physically secured is protection of proprietary software. Kent developed a model of a node that provides such protection through a combination of tamper-

resistant hardware and encryption protocols [KENT 80b].

4. Programming Languages for Distributed Applications

In order to make development of distributed applications practical, it is necessary to have a suitable programming support: a high-level programming language integrated with the operating system. Such an integration is desirable and has been applied in the context of single processor systems [LAUE 79], but is even more important in the context of distributed systems, if the application programmers are to have control over performance and reliability of their systems.

Several recent languages (e.g., Concurrent Pascal [BRIN 75] or Modula [WIRT 77]) have features for concurrency, but they assume that processes interact through shared memory. The language PLITS [FELD 79] is designed for distributed processing, but it does not place strong emphasis on integrating the language and operating system features. At the Laboratory for Computer Science at MIT, we have been developing a *programming system* for implementation of distributed applications that embody the notion of autonomy [SVOB 79a, LISK 79]. The programming system is envisioned as a set of tools that include primitives found in conventional higher level languages such as Pascal or PL/1, but also primitives normally assumed to be part of an operating system. The primary design goals for this programming system and the related research are summarized below:

Support well-structured programming. It has been successfully demonstrated that powerful abstraction mechanisms, and specifically data abstractions, aid in the production of well-structured programs [LISK 77a, WULF 76]. Data abstractions are believed to be very important to design of distributed applications, because they allow the builder of a distributed application to work with application-specific entities, without concern for the idiosyncracies of the individual autonomous nodes. Data abstractions are also convenient for hiding distributed nature and replication of databases, if it is necessary to preserve some consistency constraints on such a database.

Support communication in terms of abstract objects. Communicating program units should be able to exchange messages that contain objects meaningful at that level, rather than having to translate such objects into strings of bits deliverable by the communication subsystem. That is, the translation of abstract objects into such low level messages should be automatic. Because of the node autonomy, abstract objects may be implemented differently on different nodes. Thus, one problem is to find a common representation for various data abstractions and a mapping for each local representation to and from this common representation [HERL 80]. The second problem is a conversion of this common representation, which may be a complex data structure, into a consequtive bit stream and vice versa [SOLL 79].

Allow explicit control of the application distribution. Since many placement decisions will be based on non-technical factors external to the system, the application builders have to be

aware of the distributed nature of the system, and have the power to determine where parts of the application will reside; this decision should not be arbitrarily changed by the operating system in order to improve performance. A special abstraction called a *guardian* [SVOB 79a, LISK 79] was proposed as the basic building block (module) of distributed programs. A guardian is in fact an abstract node: it has its own local memory that is not directly accessible to other guardians, one or more processes that operate on the local objects, and one or more ports for communication with other guardians. A distributed application is then represented by an abstract network of guardians. The physical distribution of an application is controlled by assigning the whole guardians to appropriate physical nodes.

Although several guardians may reside on the same physical node, guardians can communicate only by sending messages. The possible programming primitives for inter-guardian communication vary from a remote procedure call to general send/receive primitives that allow concurrent execution in both the sender and the receiver [LISK 79]. It has been argued that these two kinds of primitives are functionally equivalent [LAUE 78], but for each one there are situations where that kind of primitive seems more appropriate (it is easier to accomplish the desired effect) than the other one. Thus, just as common programming languages support, for example, more than one control abstraction for iteration, a programming language for distributed computing should offer *both* remote procedure calls and send/receive primitives.

Support sharing and long-term storage of information. The guardians include objects that have permanent quality that is, that exist outside of the procedure and the process that created them. Rather than storing such objects in a separate file system, long-term storage should be provided by the programming system. The programming system must also support sharing of information represented as objects that reside at different nodes and belong to different users. An important aspect of sharing is proper synchronization and access control. The mechanisms developed by Reed can make much of the needed synchronization transparent to the programmers [REED 78]. Very reliable and secure shared data storage can be provided by special servers [REED 80, SVOB 80].

Support reliable (robust) operations. Requests and responses in a distributed system must be tested for integrity and authenticity, using a combination of the built-in system features and application dependent procedures. A good programming language should provide exception handling mechanisms [GOOD 75, LISK 77b]. Exception handling that involves concurrent processes also has been investigated [LEVI 77]. In a distributed system, exception handling may generate *unsolicited messages*, that is, "response" messages that were not triggered by an explicit request [ISRA 78]. Additional facilities needed for construction of robust distributed programs are primitives for setting (extending, cancelling) timeout, checkpointing primitives, mechanisms that facilitate detection of duplicate requests

and mechanisms for selective control of recovery.

The programming system must also include tools for debugging, maintenance, and evolution of the application software. Mechanisms needed for crash recovery of individual guardians are believed to provide a suitable base for selective modification of guardians: the old guardian is forced to crash, and the new version is installed as part of the recovery. Since the whole application network of guardians can be installed on the same physical node without any changes (given that such a node has enough capacity), a lot of debugging can be done locally. However, to be able to assess the performance and robustness of a distributed application, the programming system would have to be aided by a simulator.

Summary

This paper looked at selected issues that arise in connection with a design of an operating system and programming support for distributed systems where the individual nodes operate in a highly autonomous way. A major problem in this kind of work is the lack of understanding of the applications. Thus studies of relevant application areas and trial implementations are an important step towards a more sound specification of the facilities needed for distributed processing. And, there are various related issues: design of the hardware base, that is, of the communication network and the individual nodes; design of low level protocols; and performance measurement and analysis. In conclusion, design of distributed systems is a rich problem area and much research is still needed.

References

ABRA 80 Abraham, S.M., Dala, Y.K., "Techniques for Decentralized Management of Distributed Systems," Digest of Papers, COMPCON Spring 80, San Francisco, California, February 1980, pp. 430-437.

ALSB 76 Alsberg, P.A., "A Principle for Resilient Sharing of Distributed Resources," *Proc. of the International Conference on Software Engineering*, San Francisco, California, October 1976, pp. 562-570.

BISH 77 Bishop, P., "Computer Systems with a Very Large Address Space and Garbage Collection," MIT Laboratory for Computer Science Technical Report No. 178, Cambridge, Massachusetts, May 1977.

BRIN 75 Brinch Hansen, P., "The Programming Language Concurrent Pascal," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, 1975.

CLAR 80 Clark, D.D., Svobodova, L., "Design of Distributed Systems Supporting Local Autonomy," Digest of Papers, COMPCON Spring 80, San Francisco, California, February 1980, pp. 438-444.

DOLI 77 D'Oliveira, C.R., "An Analysis of Computer Decentralization," MIT Laboratory

for Computer Science Technical Memo No. 90, Cambridge, Massachusetts, October 1977.

ELLI 80 — Ellis, C.A., Nutt, G.J., "Office Information Systems and Computer Science," *Computing Surveys*, Vol. 12, No. 1, March 1980, pp. 27-60.

ESWA 76 — Eswaran, K.P., et al., "A Model of Recoverability in Multilevel Systems," *Comm. of the ACM*, Vol. 19, No. 11, November 1976, pp. 624-633.

FARB 72 — Farber, D.J., Larson, K.C., "The Structure of the Distributed Computing System -- Software," *Proc. of the Symposium on Computer-Communication Networks and Teletraffic*, Brooklyn, New York, April 1972, pp. 539-545.

FELD 79 — Feldman, J.A., "High Level Programming for Distributed Computing," *Comm. of the ACM*, Vol. 22, No. 6, pp. 353-367.

GARC 79 — Garcia-Molina, H., "Performance of Update Algorithms for Replicated Data in a Distributed Database," Stanford University Department of Computer Science Report No. STAN-CS-79-744, Stanford, California, June 1979.

GIFF 79 — Gifford, D.K., "Weighted Voting for Replicated Data," *Proc. of the ACM/SIGOPS Seventh Symposium on Operating Systems Principles*, Asilomar, California, December 1979, pp. 150-162.

GOOD 75 — Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation," *CACM*, Vol. 18, No. 12, December 1975, pp. 683-696.

GRAY 78 — Gray, J.N., "Notes on Data Base Operating Systems," *Lecture Notes in Computer Science*, Vo. 60, Springer-Verlag, New York, 1978, pp. 393-481.

GRAY 79 — Gray, J., et. al., "The Recovery Manager of a Data Management System," IBM Research Laboratory Technical Report RJ2623, San Jose, California, August 1979.

HAMM 79 — Hammer, M., Shipman, D., "Reliability Mechanisms in SDD-1: A System for Distributed Databases," Computer Corporation of America and MIT, Cambridge, Massachusetts, July 1979.

HERL 80 — Herlihy, M.P., "Transmitting Abstract Values in Messages," MIT Laboratory for Computer Science Technical Report No. 234, Cambridge, Massachusetts, April 1980.

ISRA 78 — Israel, J.E., Mitchell, J.G., Sturgis, H.E., "Separating Data from Function in a Distributed File System," *Proc. of Second International Symposium on Operating Systems*, IRIA, October 1978.

KENT 76 — Kent, S.T., "Encryption-Based Protection Protocols for Interactive User-Computer Communication," MIT Laboratory for Computer Science Technical Report No. 162, Cambridge, Massachusetts, May 1976.

KENT 80a — Kent, S.T., "Security Requirements and Protocols for a Broadcast Scenario," *IEEE Transactions on Communications* (Special Issue on Network Security), 1980.

KENT 80b — Kent, S.T., "Protecting Externally Supplied Software in Small Computers," MIT Laboratory for Computer Science Technical Report, Cambridge, Massachusetts, Fall 1980 (to be published).

LAMP 79 Lampson, B.W., Sturgis, H.E., "Crash Recovery in a Distributed Data Storage System," Xerox Palo Alto Research Center, Palo Alto, California, April 1979, to be published in *Comm. of ACM.*

LAUE 78 Lauer, H.C., Needham, R., "On the Duality of Operating System Structures," *Proc. Second International Symposium on Operating Systems,* IRIA, October 1978.

LAUE 79 Lauer, H.C., Satterthwaite, E.H., "The Impact of Mesa on System Design," *Proc. of 4th International Conference on Software Engineering,* Munich, Germany, September 1979, pp. 174-182.

LESS 79 Lesser, V.R., Corkill, D.D., "Functionally-Accurate Cooperative Distributed Systems," COINS Technical Report 79-12, University of Massachusetts at Amherst, Amherst, Massachusetts, February 1979.

LEVI 77 Levin, R., "Program Structures for Exceptional Condition Handling," Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, June, 1977.

LEVI 79 Levin, R., Schroeder, M.D., "Transport of Electronic Messages Through a Network," Xerox Palo Alto Research Center Technical Report CSL-79-4, Palo Alto, California, April 1979.

LIND 79 Lindsay, B.G., et al., "Notes on Distributed Databases," IBM Research Laboratory Technical Report No. RJ2571, San Jose, California, July 1979.

LISK 77a Liskov, B., et al., "Abstraction Mechanisms in CLU," *Comm. of the ACM,* Vol. 20, No. 8, August 1977.

LISK 77b Liskov, B., et al., "Structured Exception Handling: Issues and a Proposed Notation," M.I.T. Laboratory for Computer Science, Computer Structures Group, CSG Memo 155, December 1977.

LISK 79 Liskov, B., "Primitives for Distributed Computing," *Proc. of the ACM/SIGOPS Seventh Symposium on Operating Systems Principles,* Asilomar, California, December 1979, pp. 33-42.

MENA 78 Menasce, D.A., et al., "A Locking Protocol for Resource Coordination in Distributed Databases," *Proc. of ACM-SIGMOD Conference on Management of Data,* Austin, Texas, May 1978.

MONT 78 Montgomery, W.A., "Robust Concurrency Control for a Distributed Information System," MIT Laboratory for Computer Science Technical Report TR-207, Cambridge, Massachusetts, December 1978.

MONT 79 Montgomery, W.A., "Polyvalues: A Tool for Implementing Atomic Updates to Distributed Data," *Proc. of the ACM/SIGOPS Seventh Symposium on Operating Systems Principles,* Asilomar, California, December 1979, pp. 143-149.

NEED 78 Needham, R.M., Schroeder, M.D., "Using Encryption for Authentication in Large Networks of Computers," *Comm. of ACM,* Vol. 21, No. 12, December 1978, pp. 993-999.

PAXT 79 Paxton, W.H., "A Client-Based Transaction System to Maintain Data Integrity," *Proc. of the ACM/SIGOPS Seventh Symposium on Operating Systems Principles,* Asilomar, California, December 1979, pp. 18-23.

REDE 79 Redell, D.D., et al., "Pilot: An Operating System for a Personal Computer," *Comm. of ACM*, Vol. 23, No. 2, February 1980, pp.81-92.

REED 78 Reed, D.P., "Naming and Synchronization in a Decentralized Computer System," MIT Laboratory for Computer Science Technical Report 205, Cambridge, Massachusetts, September, 1978.

REED 79 Reed, D.P., "Implementing Atomic Actions on Decentralized Data," presented at the ACM/SIGOPS Seventh Symposium on Operating Systems Principles, Asilomar, California, December 1979; submitted to *Comm. of ACM*.

REED 80 Reed, D.P., Svobodova, L., "SWALLOW: A Distributed Data Storage System for a Local Network," submitted to the International Workshop on Local Networks to be held in Zurich, Switzerland, August 1980.

SALT 74 Saltzer, J.H., "Protection and Control of Information Sharing in Multics," *Comm. of ACM*, Vol. 17, No. 7, July 1974, pp. 388-402.

SALT 78a Saltzer, J.H., "Research Problems of Decentralized Systems with Largely Autonomous Nodes," *ACM Operating Systems Review*, Vol. 12, No. 1, January 1978, pp. 43-52.

SALT 78b Saltzer, J.H., "Naming and Binding of Objects," *Lecture Notes in Computer Science*, Vo. 60, Springer-Verlag, New York, 1978, pp. 99-208.

SMIT 79 Smith, R.G., "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver," *Proc. of 1st International Conference on Distributed Computing Systems*, Huntsville, Alabama, October 1979, pp.185-192.

SMIT 80 Smith, R.G., Davis, R., "Frameworks for Cooperation in a Distributed Problem Solver," *IEEE Trans. on SMC*, to appear.

SOLL 79 Sollins, K.R., "Copying Complex Structures in a Distributed System," MIT Laboratory for Computer Science Technical Report No. TR-219, Cambridge, Massachusetts, September 1978.

STON 78 Stonebraker, M., "Concurrency Control and Consistency of Multiple Copied of Data in Distributed INGRES," *Proc. of Third Berkeley Workshop on Distributed Data Management and Computer Networks*, San Francisco, California, August 1978, pp. 235-258.

STRO 79 Stroustrup, B., "Communication and Control in a Distributed Computer System," PhD. Thesis, University of Cambridge, England, February 1979.

SVOB 79a Svobodova, L., Liskov, B., Clark, D., "Distributed Computer Systems: Structure and Semantics," MIT Laboratory for Computer Science Technical Report No. TR-215, Cambridge, Massachusetts, March 1979.

SVOB 79b Svobodova, L., "Reliability Issues in Distributed Information Processing Systems," *Proc. of the Ninth IEEE Fault Tolerant Computing Symposium*, June 1979, pp.9-16.

SVOB 80 Svobodova, L., "Management of Object Histories in the SWALLOW Repository", MIT Laboratory for Computer Science Technical Report, Cambridge, Massachusetts, to be published.

SWIN 79 Swinehard, D., McDaniel, G., Boggs, D., "WFS: A Simple Shared File System for a Distributed Environment," *Proc. of the ACM/SIGOPS Seventh Symposium on Operating Systems Principles*, Asilomar, California, December 1979, pp. 9-17.

TAKA 79 Takagi, A., "Concurrent and Reliable Updates of Distributed Databases", MIT Laboratory for Computer Science Technical Memo No. 144, Cambridge, Massachusetts, November, 1979.

THOM 79 Thomas, R.H., A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, Vol. 4, No. 2, pp. 180-209.

TRAI 78 Traiger, I.L., et al.,"Transactions and Consistency in Distributed Database Systems," IBM Research Laboratory Technical Report RJ2555, San Jose, California, June 1979.

WILK 80 Wilkes, Needham, "The Cambridge Model Distributed System," *ACM Operating Systems Review*, Vol. 14, No. 1, pp. 21-29.

WIRT 77 Wirth, N., "Modula: A Language for Modular Multiprogrmming," *Software Practice and Experience*, Vol. 7, No. 1, January 1977.

WULF 76 Wulf, W.A., et al., "An Introduction to the Construction and Verification of Alphard Programs, *IEEE Transactions on Software Engineering*, Vol;. SE-2, No. 4, December 1976.

WYLE 79 Wyleczuk, R., "Timestamps and Capability-Based Protection in a Distributed Data Base Environment," MIT Laboratory for Computer Science Technical Memo No. 135, Cambridge, Massachusetts, June 1979.