Distributed System Software Design Paradigm with Application to Computer Networks

NORMAN F. SCHNEIDEWIND, SENIOR MEMBER, IEEE

Abstract—A paradigm for the system and software design of distributed systems is presented with application to an actual large scale computer network involving both local area networks and a wide area network. A number of design principles are offered with particular reference to how they can be applied to the design of distributed systems. The major contribution of this paper to the field of distributed systems is an explanation of how to make design decisions about distributed systems in a way which will enhance maintainability and understandability of the software and, at the same time, result in good system performance. Our aim is to recognize the implications for software quality of various decisions which must be made in the process of specifying a distributed system.

Index Terms—Computer networks, design decisions, design paradigm, distributed systems, software design.

I. INTRODUCTION

T HE United States Navy's Stock Point Logistics Integrated Communications Environment (SPLICE) is a distributed computer network for providing high speed logistics transaction processing to support the Navy's worldwide material requirements. The system consists of local computer networks and an interconnecting wide area network: the Defense Data Network. Ultimately, there will be 62 local networks world-wide—one at each major Navy and Marine Corps supply base. When complete, Splice will be one of the world's largest computer networks. Fig. 1 is provided to give the reader an overall picture of the network configuration of SPLICE. It shows both local area networks and the Defense Data Network (DDN) along with the use of datagram service (best effort delivery) on network segments and a virtual circuit service, end-to-end, for reliable user process-to-user process message delivery. A detailed explanation of the protocols shown on this diagram is beyond the scope of this paper.

The purpose of this paper is to present a paradigm for the system design and software engineering of distributed systems. The paradigm is illustrated by showing how it was used to design and implement SPLICE. The major contribution of this paper to the field of distributed systems is to explain how to make design decisions about distributed systems in a way that will provide the following: 1) maintainable (reduce the ripple effect of software changes) and understandable software, and 2) good system performance. The key to achieving the first objective is to structure the system so that software modules are essentially independent of each other (i.e., in order to perform its functions no module requires detailed knowledge of other modules' functions and information base in order to perform its functions) [9]. The strategy for achieving the second objective is to simplify communication protocols so that both the quantity and complexity of information exchange is minimized. The elegance of this approach is that the objectives are mutually reinforcing. That is, a reduction in control information exchange contributes to module independence and, conversely, efforts to increase module cohesion and decrease coupling (i.e., module independence) [16] reduces the amount and complexity of overhead traffic in a network which, in turn, improves performance.

A complex system, like a distributed computer network, requires a model for identifying, relating, structuring, and controlling system objects. If this approach is not used, the system and software design process becomes driven by ad hoc design decisions with resulting confusion concerning the interpretation and understanding of the design at any point in the process.

The paradigm provides a unified approach to distributed system design by integrating existing concepts with new concepts proposed in this paper. The following is a list of these concepts.

Existing Concepts:
- Use named objects to represent system and software entities.
- Associate a major function with each object.
- Design general functional modules which can serve a variety of applications and put unique application functions in application modules.
- Use a message form of communication to allow independence of module operation.
- Access objects by name and not by physical address.
- Put exception handling in interface modules.
- Prevent deadlock by using a dedicated resource allocation strategy.

(This concept supports the implementation of the other concepts but is not directly related to them.)

Proposed Concepts:
- Use a logical bus communication concept on local networks for maximizing accessibility (i.e., direct communication between objects) and for flexibility of using object names for message communication.)
Reduce control message (e.g., handshaking) communication to the minimum by using an interrupt driven message recognition system.

Use system state information as soon as it is available in order to eliminate the module coupling which would be involved if the system were designed to recapture the information later.

Design objects to always respond the same way to the same inputs even though operating conditions change. Accommodate differences in performance and operating conditions by placing the exception logic in interface modules.

The paradigm is based on the following general design principles, which are directly related to the above list of concepts:

- Use of objects as a means of representing system entities so that the representation of resources and processes can be completely general and become specific only at the point of naming, addressing, or invoking. This method of representation provides for the latest possible binding of objects to physical resources and for mobility of objects across physical nodes, since the identification of entities is not tied to physical nodes nor to host addresses. Objects provide a natural representation for entities in model building. They provide the basis for a virtual machine that can be used as a computer-based executable instance of a corresponding entity in the user's problem domain. The object oriented design treats functions and data as two indivisible aspects of objects in the problem domain. In this approach the problem domain is examined, the objects and their behaviors in the domain are identified, and decisions are made as to how the behaviors will be implemented in the computer. As a consequence, the designer is not forced to state the problem in computer-domain terms [6].

The object approach to distributed system design allows the problem to be decomposed into a few succinctly described parts as follows.

- What objects are required?
- What should they be named?
- How should they be located and accessed?
- How should they communicate?
- How should they be controlled and coordinated?
- Specification of generalized and independent functional modules (i.e., server modules, such as terminal management) for performing all nonunique application processing in order to:
  1. reduce redundancy of effort and documentation in module development,
  2. reduce the redundancy of storage space and execution time during network operation, and
  3. confine the effects of changes to the software to a small number of standardized modules.

This approach is important in distributed system design because of the large geographic dispersion (e.g., 62 SPLICE locations) of distributed systems.

- Use of location-independent access to objects so that users are not constrained in their access to resources by host identity or geographic location. Instead, users are free to access resources by type of service.

- Minimization of module interaction (i.e., elaborate handshaking message exchanges) by using message communication instead of the remote procedure call (RPC) so that reduced software development and maintenance effort and decreased computer execution time and communication channel transmission time will result from a reduction in the number and types of messages flowing in the network. This result follows from the fact that a reduction in number of control messages will reduce overhead time. In addition, a reduction in types of control messages will reduce software development and subsequent maintenance. The RPC, with its request-result characteristic, is inefficient for broadcast and multicast communication [17], simplex (one way) communication.
file transfer [3], and $N$ messages outstanding (unacknowledged) protocols. In these cases, RPC would require more interactions between client and server than would message passing. When message passing is used with broadcast and multicast communication, a broadcast code or distribution list can be used, respectively, to reach multiple addresses with a single transmission. Simplex transmission can be performed with no binding between sender and receiver (no result is expected); a special case of this mode is file transfer. Finally, there are many instances when it is desirable to send $N$ messages continuously (to avoid delays in the telephone network that would be incurred with a stop and wait acknowledgment procedure) and to acknowledge the $N$ messages with a single acknowledgment; a request-result procedure is not applicable.

- Use of system state information as soon as it becomes available for the purpose of planning resource utilization as far in advance as possible in the execution of system operations so that unnecessary message exchange and module interaction can be avoided. We confine knowledge of transaction state information (i.e., status and progress of a transaction) to a single module. By this method we hide major design decisions [8] and, in addition, hide important control information during system execution, as well. Section VIII-B provides details on how state information is exchanged among objects.

- Management of objects so that they always respond the same way to the same stimuli. That is, an object should not respond one way at one time to a given input (e.g., a message to be processed FIFO) and another way at another time to the same input (e.g., same message to be priority processed), even though network operating conditions may have changed (e.g., congestion in the network). Any differences in instantiations of the objects, given the same inputs, requires complex program development and maintenance and does not allow for code sharing. In the example, if the message is handled in a special way due to performance considerations (e.g., priority message handling), this capability should not be incorporated in the many server modules of the network. Rather, the exception handling conditions should be placed in a single interface module. If this is not done, exception handling conditions would have to be incorporated in every module rather than in one module. If exception handling changes in the future, many modules would have to be changed rather than one.

Table I shows the progression from the highest level of

---

**TABLE I**

**RELATIONSHIP AMONG DESIGN FUNCTION, DESIGN PRINCIPLE, AND DESIGN TASK**

<table>
<thead>
<tr>
<th>Design Function</th>
<th>Design Principle</th>
<th>Decomposition of Design Principle into Design Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identification of Objects, Functions, and Services</td>
<td>Objects Represent System Entities</td>
<td>• Postulate Set of Objects and Associate with Functions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Specify Objects and Functions by Service Performed</td>
</tr>
<tr>
<td>Identification of Functional Modules</td>
<td>Generalized and Independent Functional Module Specifications</td>
<td>• Separate Software Architecture from Software Design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Design for Multiple Use of Modules</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Specify Independent Modules</td>
</tr>
<tr>
<td>Naming and Addressing of Objects</td>
<td>Location-Independent Object Access</td>
<td>• Make Object Access Independent of Location</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Make Objects Accessible by Type of Service</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Provide Global Name Space with Mapping to Local Names</td>
</tr>
<tr>
<td>Interprocess Communication</td>
<td>Minimum Module Interaction Overhead</td>
<td>• Hide Knowledge of Message Communication from Objects</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Provide Direct Communication Between Objects on a Logical Bus</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Separate Control Data from User Data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Minimize Control Message Exchange Between Objects</td>
</tr>
<tr>
<td>Object Control</td>
<td>State Information Used Early for Assignment of Resources Consistent Object Response to the Same Stimuli</td>
<td>• Centralize Service Functions to Capture State Information</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Allow Any Number and Sequence of Object Invocations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Centralize Exception Handling in Interface Module</td>
</tr>
</tbody>
</table>
abstraction of design function, through the application of design principles, and to the breakdown of design principles into tasks.

II. ORGANIZATION OF PAPER

Topics are covered in the following sequence:

- Definitions.
- Identification of objects, functions, and services.
- Identification of functional modules.
- Naming and addressing of objects.
- Interprocess communication.
- Object Control (i.e., sequencing, prioritizing, monitoring, reporting, communicating, and preventing deadlock).
- Conclusion.

Examples are given of the application of the design paradigm to SPLICE.

Important aspects of the major topics which were considered are the following:

- Distributed system design issues.
- Our approach to resolving these issues and discussion of alternate approaches.
- Recommended design methodology for distributed systems.

Although important considerations, the following topics are beyond the scope of this paper and will be addressed in future research:

- Procedures for recovery from system malfunctions.
- Quantitative analysis of system performance.
- Security.

III. DEFINITIONS

Definitions of important terms used in this paper follow:

Accessibility: The number of objects which a given object can reach directly (i.e., without going through intermediate objects). This definition applies whether the objects of concern are executing or not executing.

Connectivity: The number of object pairs which are directly connected by physical links, depending upon the state of the network (i.e., whether there are failed units).


Functional: Designed for or adapted to a particular need or activity.

Functional Module (FM): An object which is functional (i.e., dedicated to a specific activity such as terminal management).

Global State: State information that is centralized in and maintained by Session Services.

Kernel: That part of an operating system which provides common services (e.g., memory allocation) in each processor.

Local State: State information that is maintained by each Functional Module.

Logical Bus: A method of communication on a LAN such that any object can communicate directly with any or all other objects, independent of the type of physical connection between objects.

National Communications (NC): Protocol conversion between LAN’s and Defense Data Network.

Node: A physical element of a network (e.g., processor) which connects two physical communication links.

Object: A thing serving as the focus of attention. An object has name, state, and attributes. It could be, for example, a process, a node, processor, module, or resource. (Note: We are not using “object” in the manner defined by the Ada language.)

Path: The physical route of a message from sender to receiver.

Port: A data abstraction associated with an object. It has an address and a buffer.

Process: An FM in execution.

Recovery Management (RM): Analysis and resolution of system errors for the purpose of restoring the system to its original state or to a degraded state.

Resource: A consumable (e.g., printer paper) or non-consumable (e.g., memory) object which is used to support the operation of FM’s.

Server: An FM which provides common services to other FM’s.

Session: All of the activity (message exchange and processing) which takes place between two or more processes for the duration of a single task (e.g., file edit).

Session Services (SS): Establishment, maintenance, and coordination of local and remote sessions.

Terminal Management (TM): Input and output formatting of user process data.

IV. IDENTIFICATION OF OBJECTS, FUNCTIONS, AND SERVICES

This first design function uses objects to represent system entities in the following ways.

A. Postulate a Set of Objects and Associate with Functions

The first step in the methodology is to postulate a set of objects, as suggested by Fig. 2 and to associate them with major functions, as indicated in Fig. 3. These diagrams are simplistic because it was intended at the outset of the project to divorce the representation of system and software functions from the details of hardware and software implementation.

B. Specify Objects and Functions by the Services they Perform

An object and its associated major function can be conveniently specified by the major service which the object performs for a process and by the relationships which exist between this object and other objects which assist in providing the service. For example, as depicted in Fig. 4, the Terminal Management functional module (object) edits and formats input and output data for a user process and has a message exchange relationship with the Session
A. Separate Software Architecture from Software Design

It is considered good design practice to separate software architecture from software design, the latter being hardware dependent [11]. This we accomplished by identifying a set of functional modules which performs the major functions in each local area network (LAN). We used the approach of identifying and specifying all functional modules, their characteristics, and communication protocols, prior to any consideration of hardware implementation. This is equivalent to specifying a virtual machine as the vehicle for achieving system requirements. The significant advantage of this approach is that concerns about hardware or software details, such as bus structure or LAN protocols, are not allowed to interfere with the thought process of identifying desirable system properties. These properties should be independent of the method of implementation. This is illustrated in Fig. 4 where the specification of functional modules and their relationships is essentially independent of any processor or communication hardware characteristics.

B. Design for Multiple Use of Modules

These days, there is great interest in the reuse of software. For example, this is a leading initiative of the Department of Defense to improve its software technology [1]. A neglected area of great potential in reusability is to design software modules that can be applied to multiple applications. Application development time is significantly reduced and resistance to the ripple effect of future software changes will be maximized by providing a set of server modules which perform all user services other than unique applications functions. Server modules which provide common services are appropriate because certain services (e.g., database management) are not application unique. Figs. 5 and 6 contrast the traditional approach to application development with our approach. Rather than complete sets of application-specific modules, as in Fig. 5, our approach produces only one set of generalized modules in which only the input and output functions are replicated n times, as shown in Fig. 6 [12].

C. Specify Independent Modules

An objective was to make the modules as independent as possible (i.e., maximize their cohesion and minimize their coupling) [16]. We contributed to cohesion by allocating a single major function or subfunction to a module, as shown in Fig. 6. Coupling was reduced by using a message communication system. Message exchange allows modules to function autonomously (a major attribute of a distributed system) because a module interaction only occurs when a message is transmitted. Even then, the receiver of the message does not depend on knowledge about the characteristics of the sending module in order to function properly. The message could arrive from any module in the system and the action and response, for given message contents, would be the same because the receiving module's behavior is governed by the contents of the message and not by the identity of the sender. Furthermore, it has been claimed that message communication improves the modularity, maintainability, and understandability of software relative to other alternatives, such as the use of shared memory [15]. Shared memory communication could have been used within single processors in each LAN. However, this approach would have resulted in greater dependence between modules and, hence, greater software complexity, because of the critical section problem which is introduced when processes communicate via shared memory [10]. That is, additional control signaling would have been required to indicate when a process has exclusive use of shared memory. In addition, for a LAN of 1 to 2 kilometers in extent, it is infeasible to use shared memory for communication between widely separated processors. More will be said about our approach to object communication under Section VII.
VI. NAMING AND ADDRESSING OF OBJECTS

A. Make Object Access Independent of Location

We were seeking the most general type of system design in order to insulate parts of the system from the effects of future change. With regard to naming, this means that the system and its users must not be locked into host and physical names and addresses as has been the case in ARPANET, for example. This design task is achieved by providing a network dictionary/directory (Dic/Dir).

B. Make Objects Accessible by Type of Service

For purposes of software maintainability and ease of use, objects should be accessible by type of service (e.g., text edit) or subject (e.g., F14 engines). The use of services should be independent of geographic or host location of the objects which support the service. When an object is bound to a host, and its address, a change in the location of the object requires a change in the commands to access the object and a change in the user’s knowledge of the object’s location (i.e., host address). Also, unless explicitly commanded by the user, he should not have to be involved in data transfer functions (e.g., file transfer) in order to access data which does not reside in his local processor. This need for generality and transparency in information access is of particular concern when networks are used by unskilled clerical workers, as is the case with SPLICE.

This design task is achieved by providing a Dic/Dir.

C. Provide Global Name Space with Mapping to Local Names

We have concluded that the best way of implementing object access by type of service, with regard to naming and directory administration, is to provide a global name space with mapping to local names in the form of a centrally administered designation, distribution and update of unique names, with the complete copy of the Dic/Dir maintained at the network administration node and subsets of Dic/Dir with relevant names, resource descriptions and network addresses at each of the LAN’s.

We specified a Session Services module to perform a variety of services for the user and the functional modules, including invoking the Dic/Dir for the user, providing task analysis and initiation, coordinating sessions, monitoring session progress, and reporting errors to the user and to the Recovery Management module. The type of service or subject terms entered by the user are given to Dic/Dir by Session Services to try to identify and locate functions or resources in the network which appear to satisfy the user’s requirement; the information is presented to the user for his action.

VII. INTERPROCESS COMMUNICATION

A. Hide Knowledge of Message Communication from Objects

Several aspects of interprocess communication which are critical to achieving a good software design will be discussed. The first of these pertains to the method by which one process (object) invokes another. If a pair of processes is tightly bound (a process has to know about the existence of the other process in order to communicate), or processes can only communicate directly with a subset of the total objects in the network, this situation makes objects special rather than general and requires this information to be stored in the appropriate objects. For example, changes to network routing tables would be fine-grained and difficult to make.

Hiding knowledge of one object from another object is accomplished by not storing name, address, function, etc.,
information in objects. Instead, one object invokes another indirectly by using the Services Code and Network Services Directory, as shown in Fig. 4.

B. Provide Direct Communication Between Objects

When other objects lie on the path between two objects which wish to communicate, routing information must be stored in the intermediate objects. This will make maintenance difficult as the network configuration changes in the future. In addition, performance will be degraded as a result of queuing delay caused by intervening objects. A method of maximizing connectivity and accessibility is to use a logical bus communication system for LAN's. The concept of objects communicating on a bus is shown in Fig. 7. Fig. 8 shows the arrangement of functional module in a bus configuration. This design has the significant advantage of an inherent ability to provide broadcast and multicast communication: a message placed on the bus can be read by all or a subset of objects; no physical address is required to transmit a message; only an object's name is required. A receiving object can either act on the message or ignore it depending upon whether its name is in the message. In certain cases, the use of broadcast communication and its associated reception of all messages by all objects may have to be tempered by security considerations. Only a minor part of the traffic in SPLICE is classified.

Assuming the selected physical design provides good performance, it is convenient to have a logical design which can be mapped directly to a physical design. Naturally, it is easy to map from a logical bus to a physical bus implementation. The relationship is less obvious, particularly for broadcast communication, when another physical configuration, such as a ring, is used. Nevertheless, the logical bus could still be used by transmitting a broadcast message (i.e., one with a code that indicates that all objects on the ring should read the message). The analysis of physical LAN alternatives involves complex performance tradeoffs and is beyond the scope of this paper.

C. Separate Control Data from User Data

A feature of the logical bus is the separation of control data from user data by using separate logical data links and ports for each. This is another aid to maintenance: changes in control data procedures and message formats will not affect user data protocols. This concept is shown in Fig. 9. Fig. 9. Object communication protocol with data and control links.

D. Minimize Control Message Exchange Between Objects

An alternative to message communication is the remote procedure call [3]. This method appears to require more explicit binding between objects than does message processing. Furthermore, because each object must be invoked separately, its accessibility is not as high where broadcast or multicast communication is involved. In our design the stimulus for activating an object is the reception of a message and not the identity of the transmitting object.

Many networks provide for elaborate handshaking or connection opening and closing protocols or for a master-slave relationship between processes [14]. Other systems cause the sending process to block while awaiting a reply from the receiving process [4]. We are not convinced that such procedures are necessary. It is certainly not desirable in military networks, such as SPLICE, where rapid message communication is paramount.

Thus, we provide for a common interrupt driven object activation method, supported by an operating system kernel (see Fig. 10), which is implemented in all modules of SPLICE. In this approach no module is "owned" by another module; each module is autonomous and can proceed to execute other primitives—both send and receive—while awaiting a reply from a previous message. This mode of operation is achieved by using interrupts to activate an object, thus providing an asynchronous relationship between objects in contrast to the more traditional
lock-step synchronous association. We do this to minimize interaction between modules so that complex software will not be required for handshaking; in addition, we improve performance by minimizing the number of control messages in the network. Our objects do not "listen" for an incoming message; they only act upon receipt of a message, which interrupts the operating system kernel which, in turn, passes control to an object, as indicated in Fig. 10. As can be seen in Fig. 10, the message processing objects, indicated by $O_i$ and $O_j$, do not perform interrupt processing. This task is performed by the operating system kernel object $O_k$. Thus the use of interrupts does not complicate the message processing software. Furthermore, interrupt processing is required in the system anyway for ordinary (nonnetwork) I/O. Network interrupts are treated like any other I/O interrupts. Message processing objects maintain information about the status of communication with other objects (e.g., expected response from another object). An inability to handle a message at the receiving object, such as a buffer full condition, would be indicated at the sending object by the expiration of a time-out or by receipt of a "reject" message from the receiver, as in the case of a message transmitted by an unauthorized sender.

We recognize that simplification achieved through reduction in handshaking may be at the expense of greater complexity at the operating system level caused by interrupt handling. However, our method will definitely improve performance because handshaking message exchange takes place across slow telephone channels, whereas interrupt handling takes place within fast processors.

Handshaking can be reduced, even in an unreliable communications environment, by using acknowledgments and timers to confirm or not confirm that communication is occurring between two objects.

The potential for an interrupting message causing deadlock by requiring resources held by the interrupted process is prevented by allocating all resources required to process the message from a resource pool (no resources held by the interrupted process are deallocated for this purpose). If the required resources are not available, the message is not accepted and retransmission will occur at the sender due to time-out on lack of acknowledgment from the receiver. (See discussion in Section IX.)

VIII. OBJECT CONTROL

A. Coordination

By object control we mean the mechanisms which are used to coordinate: 1) sequence and priority of object activities; 2) status monitoring and reporting, including error reporting; 3) object communication, including message exchange and interrupt handling; and 4) deadlock prevention.

Although SPLICE is a distributed system, certain system functions must be centralized in each LAN because they are pervasive and it would be duplicative of program storage and development effort to perform these functions in every FM. For example, it would not be efficient to have every FM deal directly with user processes, or for every FM to monitor status and report errors, or for every FM to assist the user in obtaining directory services. If module directory functions, for example, were to reside in all Functional Modules, there would be significant duplication of code and consequent waste of storage space and, more important, a change in a coordinating function, such as directory logic, would require changes in all Functional Modules because all Functional Modules would have an interface to the directory [2], [13]. By confining much of the detailed logic of object communication and coordination to an interface module—Session Services [9], the software maintenance problem is alleviated. Thus, the responsibility for coordinating the activities of each LAN and its FM’s was given to Session Services (SS). This module has relationships with the Terminal Management (TM) module, Dic/Dir, and other FM’s, as can be seen by referring again to Fig. 4, where the numbers on the diagram indicate the sequence of FM operations.

Fig. 11 shows these relationships from a slightly different perspective. Here we depict the use of broadcast communication and named objects to send a message, with the help of SS and Dic/Dir, to either a local or remote object. In the former case all objects will read the message header but only the object with the matching name will copy it. In the latter case the National Communications Module will recognize the object name as pertaining to a remote object and will forward the message across the long distance network.

B. Centralize Service Functions to Capture State Information

Because of these relationships, service functions are centralized in SS, since it acquires information about user requests prior to all other FM’s, except TM, which is limited to performing input/output editing for the user. Consequently, SS is able to: 1) coordinate the use of Dic/Dir for the user, as described in a previous section and 2) determine, to the extent possible, the task breakdown of the work to be done for the user (i.e., identity and sequence of FM’s which must be used to satisfy the user’s request). This cannot be done completely in all cases because the sequence of FM execution may be data dependent. That is, the identity of FM $(n + 1)$ may not be known until FM $(n)$ has processed its data. Nevertheless, the task information which is known at input time is both valuable and perishable so, in accordance with an earlier stated design principle, we want to capitalize on system state in-

![Fig. 10. Interrupt driven object communication procedure.](image-url)
formation at input time in order to reduce message exchange and module coupling from what it would be if all FM's had to interact with user processes and the Dict/Dir.

The FM which SS passes control to is called the controlling FM (CFM). A CFM is just an FM which happens to be the first (and perhaps only) FM in a sequence of FM's which is used in a session. The relationship between SS, CFM, and FM's is shown in Fig. 12. One of the functions of a CFM is to close the loop on the series of FM activities which constitute a session and to report a completion code to SS indicating either successful or unsuccessful completion of the session and, if the latter, reasons for the error condition.

SS is also responsible for maintaining the global state of the system to the extent that it has knowledge of system state. By virtue of providing initial task instructions to the CPM, SS is aware of the task to be performed and the necessity for CFM to report back to it when the task has been completed or to provide it with an error code, if the task cannot be completed. Beyond this degree of centralization of state information in SS, each FM must assume responsibility for maintaining its local state. This is consistent with the basic concept of a distributed system (i.e., decentralization of control). When an FM sends a task request to another FM, it must establish and maintain state information until its request is satisfied and it can respond to its invoking FM. Timers are required at both global and local levels to ensure timely response to task requests.

C. Allow Any Number and Sequence of Module Invocations

A fascinating aspect of the control problem is whether objects should be allowed to invoke other objects, via message communication, without restriction as to the number and sequence of invocations for a given session.

To appreciate the problem, consider a world-wide network (SPLICE) in which a desired resource could, theoretically, be located in any LAN on any node. The first access to an FM during a session could be directed to any node in the geographic domain of the network. If this FM requires the services of another FM to perform its service, a second invocation would result; this process could con-
continue indefinitely. Also to be recognized is the fact that the receiving FM may be in the same or a different LAN than the sending FM. Furthermore, the invoked FM's may be in the same or different processors than the user process. A final complicating factor is the possibility of \( \text{FM}(n) \) invoking \( \text{FM}(n+1) \) and the latter then invoking \( \text{FM}(n) \), or any subsequent \( \text{FM}(n+i) \) later invoking \( \text{FM}(n) \). Ideally the system must be able to handle any sequence of invocations regardless of the number of and geographic locations of the FM's involved in a session.

From the standpoints of reducing software complexity and achieving generality of software design, the best approach is to allow any possible sequence of invocations because all invocations would be done the same way. Any \( \text{FM}(n) \) could send a message to any \( \text{FM}(n+1) \) without restriction. That is, it would not be necessary to provide logic in the FM's to determine under what circumstances \( \text{FM}(n) \) could invoke \( \text{FM}(n+1) \) and under what circumstances it would have to return control to \( \text{FM}(n-1) \) (its "caller") for the purpose of terminating the use of "new" FM's in the session. This approach corresponds to one of our design principles which states: modules should always respond the same way to the same inputs, even though operating conditions are different (e.g., invocations increase to a very high number). The problem with this solution is that, with no limit on the number and sequence of invocations, the elapsed time for a user to obtain a response to a request could be quite high.

D. Centralize Exception Handling in Interface Module

Our resolution of this dilemma is to allow complete generality of invocation, as suggested by Fig. 12, but to put exception handling under the control of SS, which places the session in a deferred (background) processing category, if the completion code is not received from CFM by SS within a time established by SS. Processing for such a session will continue but not on an interactive basis. The user process will be notified of this condition by SS and it will be advised that a new session can be started and processed while waiting for the results of the deferred session. Alternatively, the user can log off and receive deferred processing results in a designated file; the user will receive a message the next time he logs on regarding the availability of deferred processing results. If the user remains logged on, results will also be placed in a file but he will be notified at his terminal when the results are available.

Considerable research needs to be done, using simulation and actual network exercises, to obtain a better understanding of the performance penalties which may be incurred when unlimited nesting of invocations is allowed in a network.

IX. DEADLOCK PREVENTION

One of the most interesting and challenging aspects of object control concerns deadlock prevention. We choose to prevent deadlock by a means which will reduce software complexity, increase performance but also increase the cost of hardware. The four necessary conditions for a deadlock to occur are the following [5].

1) Processes claim exclusive control of the resources they require (mutual exclusion condition).

2) Processes hold resources already allocated to them while waiting for additional resources (wait for condition).

3) Resources cannot be removed from the processes holding them until the resources are used to completion (no preemption condition).

4) A circular chain of processes exists in which each process holds one or more resources that are requested by the next process in the chain (circular wait condition).

Only one of the above conditions must be violated in order to prevent deadlock. Condition 1) cannot be used because we must allow a process to have exclusive use of a file or record during updating. The method chosen is to violate 2): a process will never have to wait for additional resources while holding other resources. This is accomplished by dedicating all resources (e.g., RAM and disk) to receiving \( \text{FM}(\text{FM}(R)) \) and each of its submodules which are needed to process a message from each of the sending FM's (\( \text{FM}(S) \)) with which it is currently communicating. Similarly, all resources needed by a given \( \text{FM}(S) \) to send a message to each receiving \( \text{FM}(R) \) with which it is currently communicating are dedicated. The allocated resources are associated with the message buffers of the sending and receiving modules. This association is shown in Fig. 13. The resulting software design is highly simplified because the data structure which records resource assignment for FM's need only contain a boolean variable for each FM which indicates that resources are either assigned or not assigned; there is no need to keep track of the allocation of varying amounts of various resources for each FM. This can be contrasted with the alternative of using 4), for example, where a complex linked list would be necessary for determining whether the condition holds. Alternative 2) does carry a resource wastage penalty, but in view of the low cost hardware-high cost software relationship, it is a price we gladly pay in return for a reduction in software complexity.

Furthermore, in view of rapidly declining hardware costs, the tradeoff in favor of using more hardware to prevent deadlock seems reasonable, whereas the "cost" of deadlock to the military in an important network like SPLICE would be considerable. A by-product of this strategy is the improvement in performance which results from the dedication of hardware resources.

Another method we use to improve performance is to also violate 3) by preempting a process from further use of the processor, if the amount of processor time consumed equals the maximum allowable time. This prevents a process from waiting indefinitely for the use of the processor.

X. CONCLUSION

SPLICE will be installed and will evolve over a period of many years. The success of SPLICE will depend on
many factors, including the performance and reliability of vendor hardware and operating system. Therefore we will never be able to say conclusively that the system and software design approach which has been described had a significant effect on the success (or failure) of the system. However we do know that the methodology helped us tremendously in thinking about the problems involved in designing a distributed system and in constructing understandable solutions; this was an important contribution to our effort.

ACKNOWLEDGMENT

I thank the referees for their valuable comments on the first draft.

REFERENCES


Fig. 13. Buffer allocation.