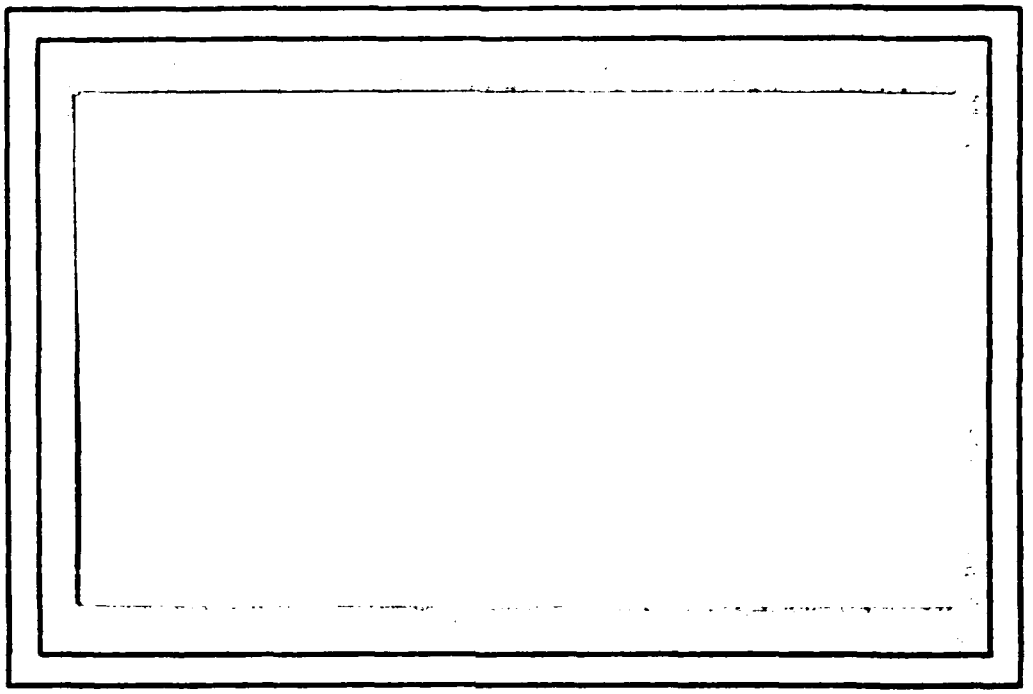
(4)



COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

4

CS-TR-2079 July, 1988

A Structuring Framework for Distributed Operating Systems *

Juergen Nehmer **

Systems Design and Analysis Group
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

This technical report is an attempt to survey the organization principles for distributed systems in a systematic and concise manner. Starting with a comprehensive set of terms covering the area of distributed computing, a classification scheme for distributed operating systems is developed. Based on this classification scheme several communication models are surveyed. Client-server models as an attractive structuring means for distributed operating systems are discussed in greater depth. The report concludes by elaborating the nature of cooperation as an unique underlying principle to organize the work in distributed systems.

---

* This work is supported in part by contract DASG60-87-C-0066 from the U.S. Army Strategic Defense Command and by contract N00014-87-K-0241 from the Office of Naval Research to the Department of Computer Science, University of Maryland, College Park, MD.

The views, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army or Navy position, policy, or decision, unless so designated by other official documentation.

** The author was on leave from Universitaet Kaiserslautern, D6750 Kaiserslautern, Fachbereich Informatik, Postfach 3049, West Germany.

1. INTRODUCTION

Distributed computing has become one of the most rapidly developing technology in the computer field in recent years. A growing number of research efforts in the computer science community are devoted to different subjects of this fascinating topic worldwide.

Distributed computing -as opposed to centralized computing-appears as an attractive alternative to configure a computing system and to organize its work. It receives its potential power from a characteristics fundamental to the structure of any distributed system: its conceptually unlimited extensibility in terms of processing nodes and storage capacity. This property gives raise to the hope that some day one will be able to built distributed systems which -in terms of processing power and reliability- will exceed by far the most powerful centralized systems ever built.

However, when more and more people were dealing with different aspects of distributed computing the terms and concepts got frequently mixed-up and misused. Today, the term "distributed computing" and all its derivates are buzzwords without any precise meaning. Compare for example the following terms and try to relate them to each other precisely:

distributed processing
distributed programming
distributed program
distributed system
distributed operating system
distributed programming language

concurrency
concurrent program
concurrent programming language

parallel processing
parallel programming
parallel program
parallel system
parallel programming language
large grain parallelism
fine grain parallelism

network

networking
network program
network operating system

It is probably hard for most readers to tell exactly the differences between a concurrent and a distributed program, or to explain how large/fine grain parallelism relates to distributed and parallel programs and so on.

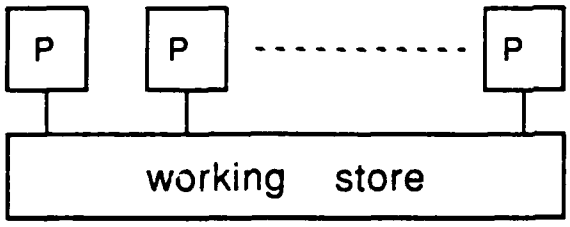
This discussion motivates our main concern followed up with this paper: instead of providing a comprehensive overview of actual research activities in the distributed operating system field as given by Tanenbaum and Renesse in their recent survey[1] we intend to focus on the development of a structuring framework for distributed operating systems. It should serve as a sufficient basis for classification and comparison of existing approaches of distributed operating systems as well as for identifying and discussing solved and unsolved problems in a systematic manner.

2. A STEP TOWARDS PRECISE TERMS

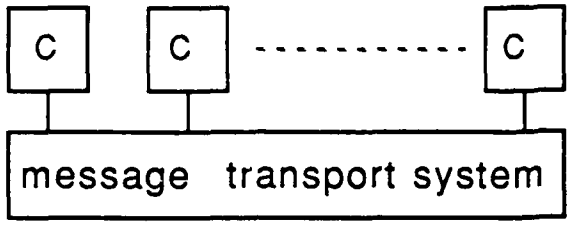
Before developing a structuring framework for distributed operating systems it is necessary to agree on a precise understanding of terms. The following proposal which is by far not comprehensive is an attempt to re-assign precise meanings to frequently used terms in the field of distributed computing(also referred to as distributed processing) and to relate them to the area of networking, concurrency and parallel processing.

To start with let us consider what the unique characteristics of distributed computing are. Loosely spoken, the term refers to the simultaneous execution of interdependent programs on a special computer architecture. So let us have a deeper look into computer architectures suitable for distributed computing and program structures adequate for simultaneous execution.

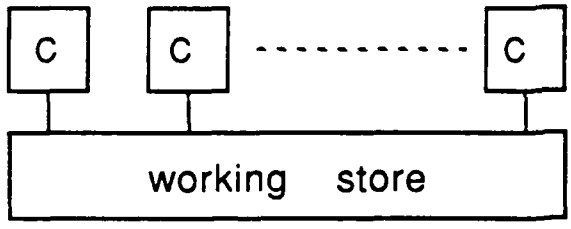
A computer capable of executing programs simultaneously is called a multicomputer. Figure 1 shows four basic multicomputer architectures. In a multiprocessor architecture(MP) as shown in



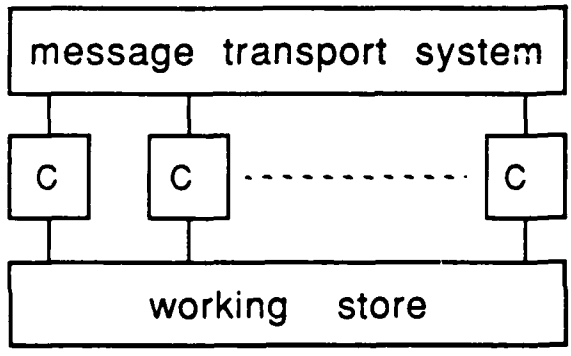
1a) Multiprocessor (MP)



1b) Message - Interconnected Multicomputer (MMC)



1c) Storage - Interconnected Multicomputer (SMC)



1d) Hybrid - Interconnected Multicomputer (HMC)

Figure 1 Basic Multicomputer Architectures

Figure 1a two or more identical processing units(P's) are hooked up to a common working store. Systems of this architecture are commercially available since the middle of the 1960's and are usually organized by a single operating system.

In a message-interconnected multicomputer architecture(MMC) as shown in Figure 1b several autonomous processor-storage units(called computer nodes) are interconnected by a message transport system. Each computer node could have local peripherals attached to it. The message transport system separates the system into spheres of independent control threads and distinct, non-overlapping address spaces. Systems of MMC-architecture are usually called computer networks.

A storage-interconnected multicomputer architecture(SMC) is shown in Figure 1c.In contrast to the previous MMC-architecture all computer nodes have access to a common storage. This architectures has been favored by data base designers since it avoids time consuming copying of bulk data between different nodes[2-6].

A combination of message -and storage interconnection can be found in hybrid- *interconnected* multicomputer architectures as for example in the Butterfly multicomputer[7].

These basic architectures can be combined in various ways to configure hierarchically and nested multicomputer systems. One attractive structure is shown in Figure 2.It consists of multiprocessors interconnected by a message transport system and combines the basic architectures of types a and b in a nested structure. Other examples are discussed in[8].A thorough match between the computer architecture and the structure of the software running on it is required in order to fully exploit the potential power of a given architecture.

A useful classification of programs with respect to distributed computing takes the number of independent threads of control and the number of disjunct address spaces as classifying parameters as depicted in Figure 3. This leads to three major program categories:

Sequential program

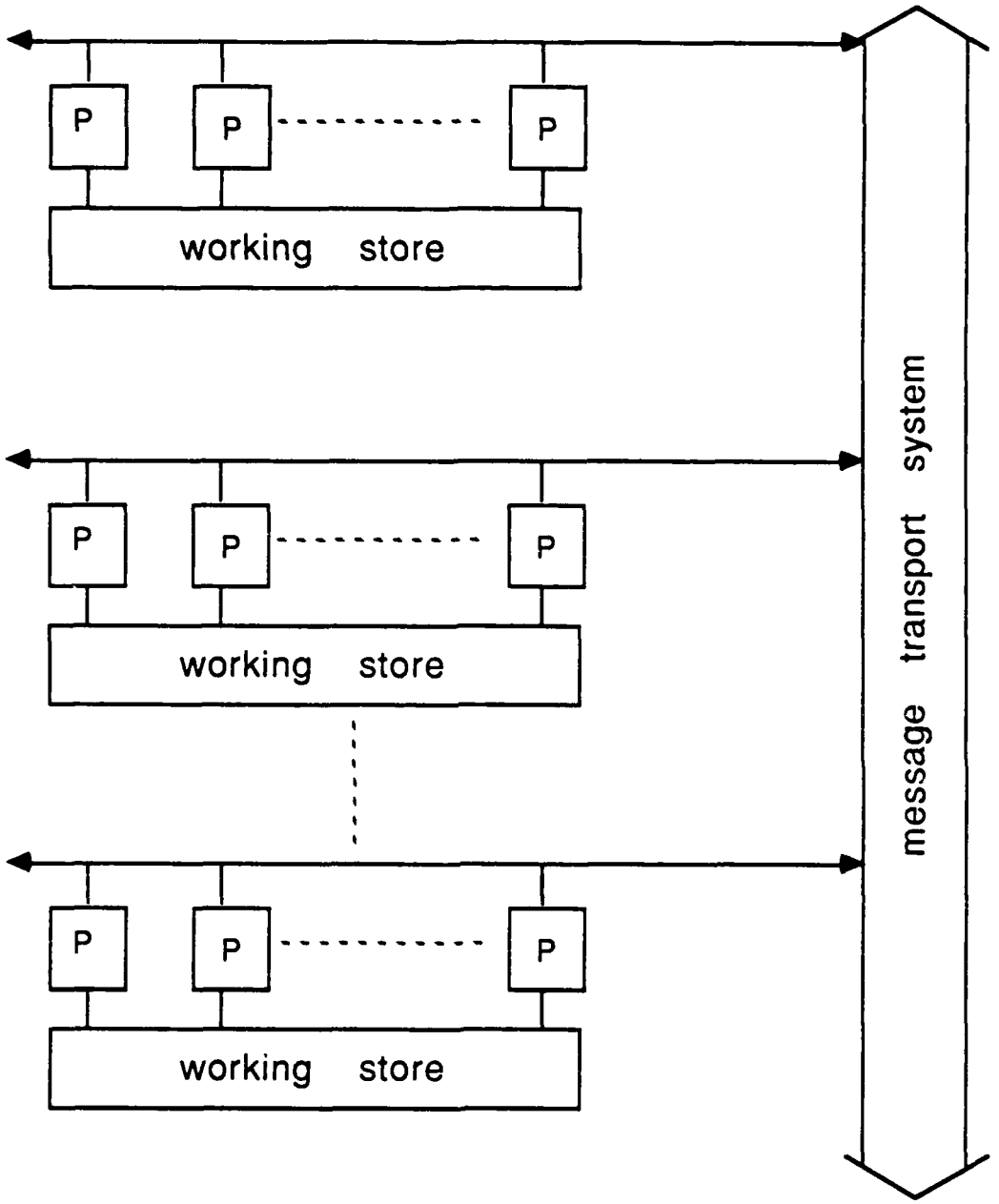


Figure 2 A nested multicomputer architecture consisting of multiprocessor systems interconnected by a message transport system

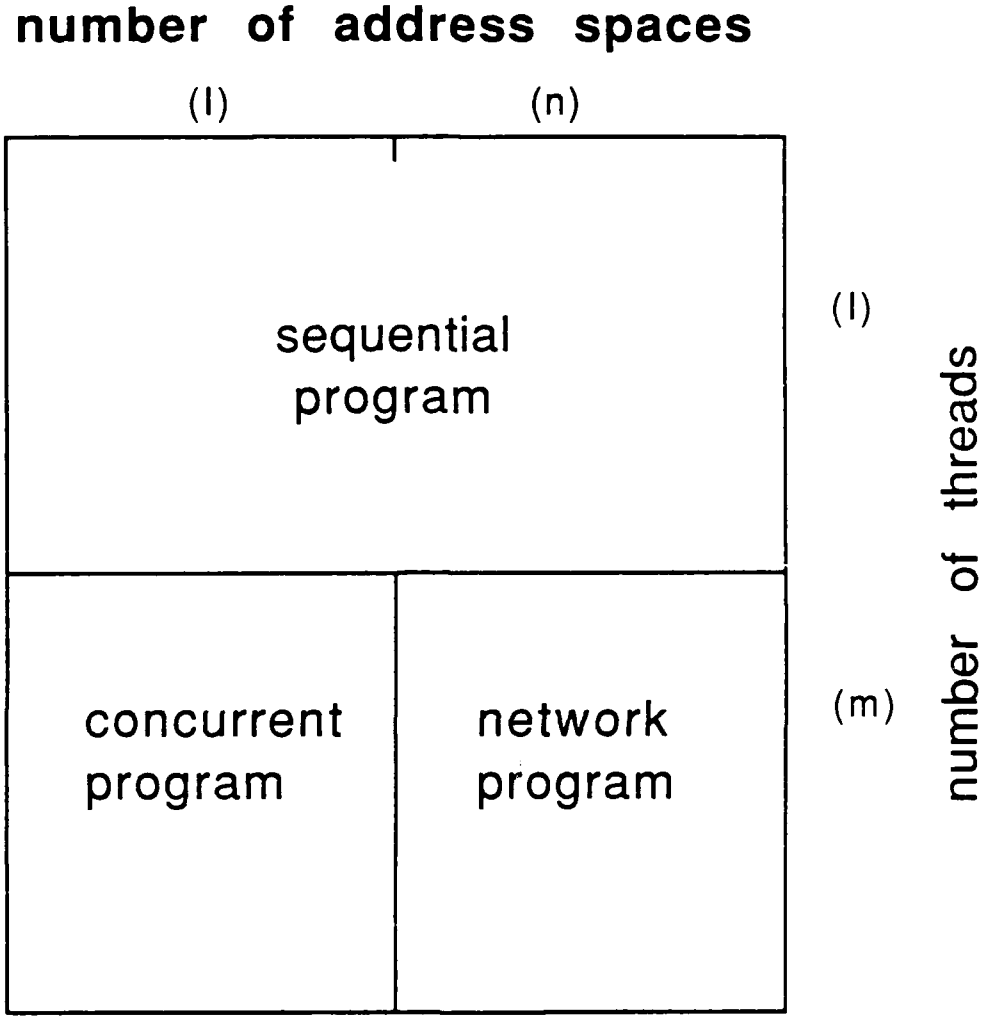


Figure 3 Classification of programs according to the number of control threads and address spaces

A sequential program is a one-thread entity defined in either one or n address spaces. The dynamic occurrence of a sequential program is called a process.

Concurrent program

A concurrent program is a multi-thread, one address space entity. The dynamic occurrence of a concurrent program is sometimes called a team(of tightly coupled processes).

Communication between processes of a team is usually achieved by giving all processes direct access to common variables. The necessary synchronization to shared variables is supported by well known mechanisms like semaphores[9]. Monitors[10] provide a more structured but indirect way of process communication within teams in that they encapsulate shared data and force processes to access them via a procedural interface. Since there is only a single address space for all processes global state information may easily be obtained from each process.

Network program

A network program is a multi-thread, multi-address space entity which comes in two variations:

-as a collection of single-thread/address space entities.

-as a collection of multi-thread/address space entities.

The coupling between the disjunct address spaces for both variations of network programs is accomplished by message exchange between programs residing in different address spaces. The dynamic occurrences of both types of network programs are called "communicating processes" or "communicating teams" respectively.

With this classification scheme for programs we are able to define a distributed program as a network program which meets the additional requirement of hiding its network structure from its users. The term "network transparency" is used here in order to denote this essential property of

distributed programs. It means that a distributed program should maintain a high degree of invisibility on its internal structure thereby making it impossible to recognize from outside on what type of computer architecture it is running.

Network transparency as defined above encompasses "location transparency" and "performance transparency". Location transparency is the invisibility of the location(from a user's point of view) where the different parts of a distributed program get executed. Performance transparency is the invisibility of performance differences in spite of the execution of parts of a distributed program at different locations(i.e. local or remote).

Notice that there exists a special case for which concurrent programs and network programs look the same: it is characterized by zero interactions between the independent threads of control. The behavior of such programs can be graphically described by a precedence graph as depicted in Figure 4. It is assumed that a fork-operation creates the independent threads of control which conceptually execute in parallel without interaction until terminated by a join-operation. We call programs with this behavior parallel programs. The parallel execution of parallel programs is best supported by array computers[1]. They are beyond the scope of this paper.

Distributed computing can now be defined as the distributed execution of a distributed program on a multicomputer. A distributed system is a permanent combination of a distributed program and a multicomputer where the principles of distributed processing apply.

Notice, that with the above definitions a sequential program running on a multicomputer is not a distributed system nor is a distributed program running on a single processor system.

It is rather easy now to relate networking to distributed computing. Networking is the more general and less restrictive term which always applies if two or more computers are linked together. A network operating system, for example, is a collection of native operating systems extended by

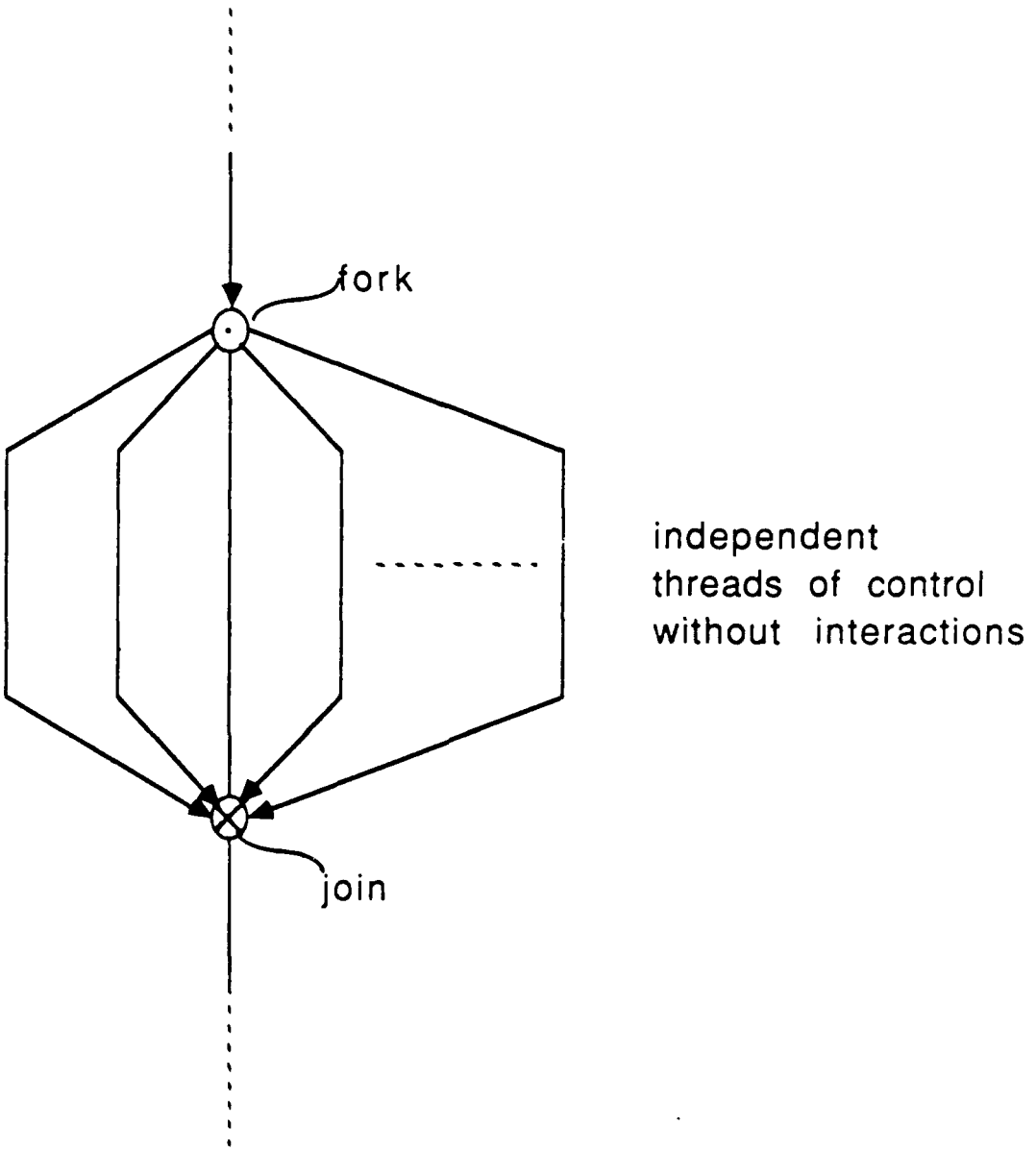


Figure 4 A precedence graph for a parallel program

mechanisms to request and receive remote logins between them. A user of a network operating system is always aware which local site he is interacting with. In contrast, a distributed operating system ideally provides a single operating system image to its users. A user should be unaware of the fact that several computers are involved in performing a requested service. Although operating systems are probably the most important class of distributed programs used to build distributed systems there is no ultimate need to base distributed systems always on the existence of a distributed operating system as Enslow did[12].Highly dedicated distributed systems as for example distributed data base systems are likely to appear which are built from scratch or on top of a primitive distributed kernel.

Having explored a more precise conception of distributed computing we will turn our attention now to the distributed programming process.

By distributed programming we simply understand the programming process which results into a *distributed program*. In order to better understand the principles which lead to some form of a distributed program structure it is helpful to start with an object-oriented view of a non-distributed program as shown in Figure 5. A program is considered there as a black box which manages some internal state S and exports the function set F for use by other programs. For the state S we assume that it can be decomposed into a set of independent data types for which several instantiations may exist. This leads to a matrix representation for S as depicted in Figure 5. A row, for example Tx, contains all instantiations of a given data type x (there may be empty matrix elements).We assume that each data type Tx is manipulated by a corresponding function set Fx. Fx is assumed to be a subset of the function set F(this is an oversimplification which may not always apply in practice but it is sufficient for the purpose of explaining the general idea behind the distributed programming process).

In the sequel we will use the term resource type instead of data type and associate with the instantiations of a given data type the number of resources available from a given resource type.

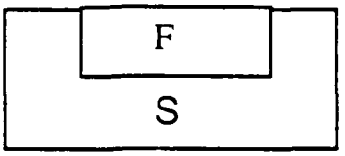
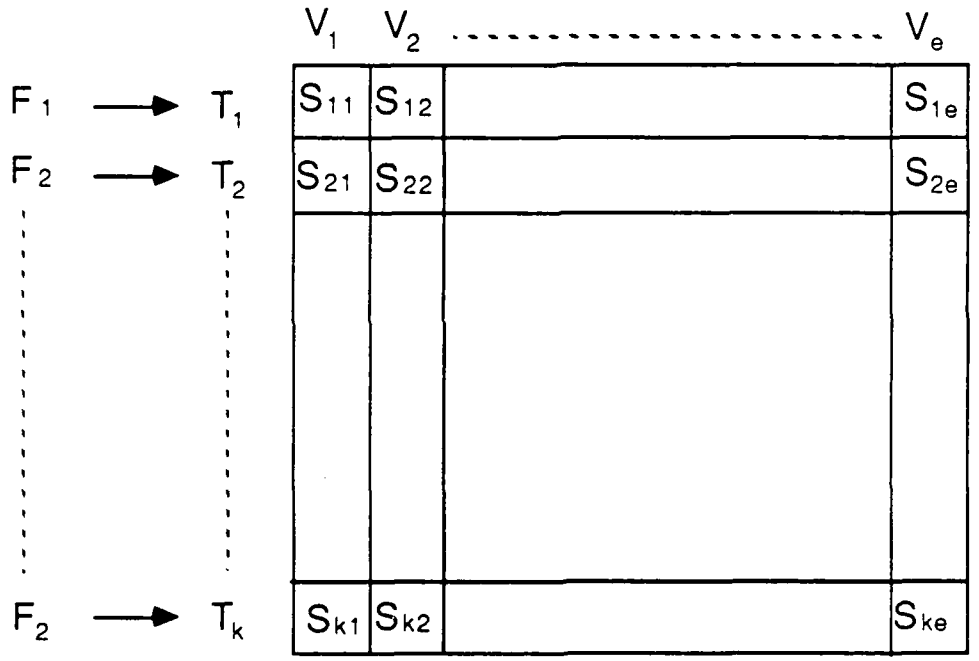


Figure 5 An objected oriented view of a program managing the state S and exporting the function set F.



T_x all data instantiations of a given data type x

V_y a collection of data instantiations from each data type

F_x operations defined for data type x

Figure 6 A data model for an object's state S

>From the refined structure of the state S one can derive three basic distribution strategies resulting in different versions of distributed program structures for our original program.

A) Full replication(Figure 7a)

In a fully replicated program the state S and the function set S are replicated n times at each node of a distributed system. This organization requires an expensive cooperation process between the identical program copies in order to maintain a consistent global state of the resources. It is the preferred method if a high degree of fault tolerance is the primary design goal for a system.

B) Function replication-resource partitioning(Figure 7b)

In this approach we partition the state S along columns of our state matrix. This leads to a partitioned program where each program copy contains the full function set of the original program but manages only a small subset of all resources.

If one node fails all others can continue operation by bypassing the resources of the failed node This organization is favorable in case that graceful degradation is the primary design goal of a system. It also yields the advantage of high uniformity of the overall system structure since all copies of the distributed program are basically identical(they differ only in the number of resources they manage=data structures).

C) Function and resource partitioning(Figure 7c)

In this approach the state S is partitioned along rows of the state matrix. This automatically leads to a function partitioning since each row is associated with a certain resource type and with a subset of the function set F therefore.

This decomposition strategy results in distributed systems with specialized nodes, each managing all resources of a given type and favors such desirable properties as isolation(of different subsystems),simplicity(because of its dedicated nature) and efficiency(because of its simplicity).

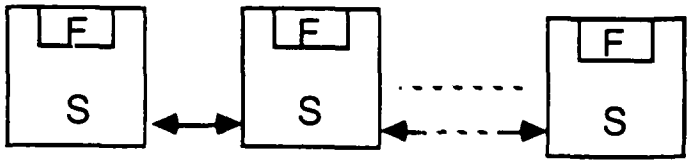


Figure 7a:
Full replication

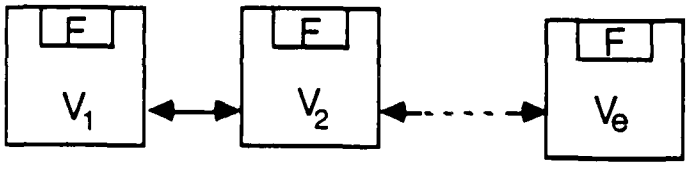


Figure 7b:
Function replication-
resource partitioning

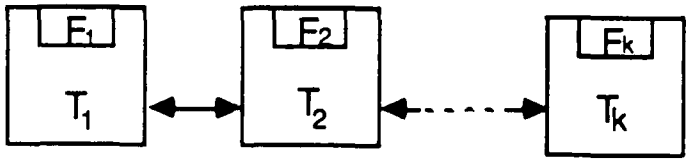


Figure 7c:
Function and
resource partitioning

Figure 7 Distribution strategies for a program described by (F,S)

The basic distribution strategies explained above can be combined in various ways in order to obtain distributed systems which combine the advantages of the different approaches. For example, a distributed operating system can be designed along the distribution strategy C above where some of its critical components may be replicated in order to provide a high degree of fault tolerance.

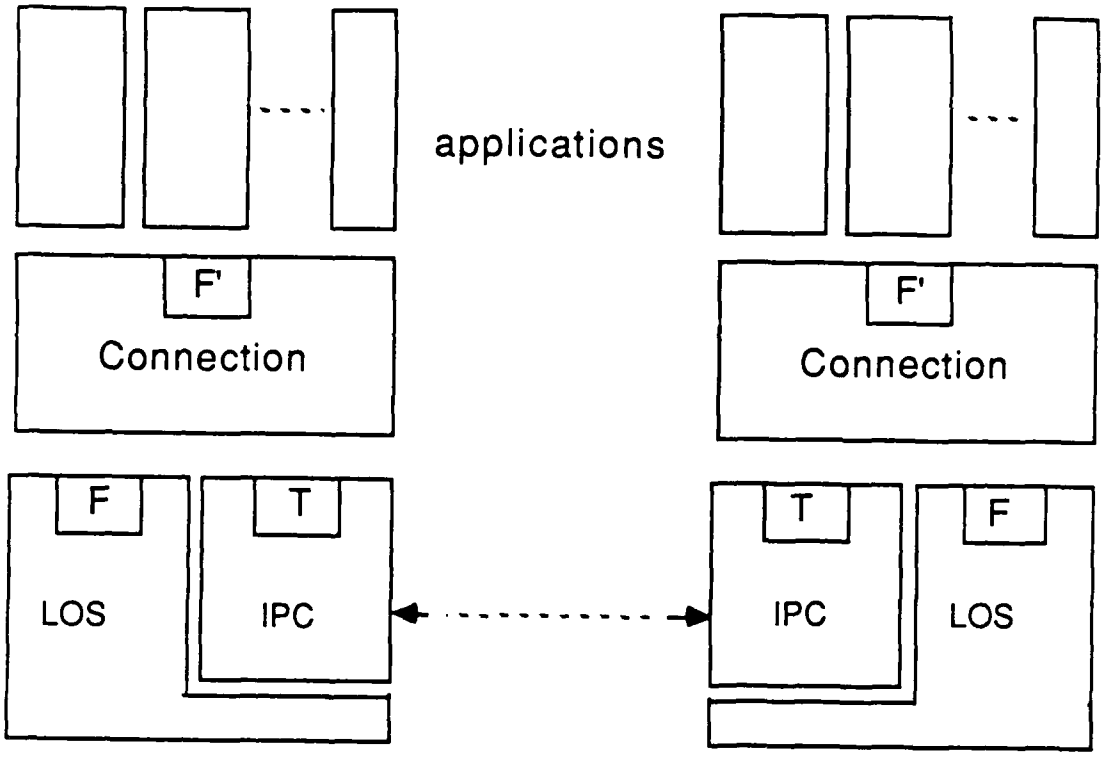
3. CLASSIFICATION OF DISTRIBUTED OPERATING SYSTEMS

According to the distribution strategies developed in the previous chapter we can identify two different classes of distributed operating systems currently in use:

A. MIDOS-Architecture

The MIDOS-architecture(Multi-Instance Distributed Operating System) follows basically the distribution strategy B as explained in the previous chapter. There the distributed operating system is made up of n identical full- function copies of the original single node operating system where each copy manages only a subset of the available resources. Figure 8 provides an overview of the resulting system architecture. As a first step the LOS's have to be extended by a inter-process communication mechanism to facilitate message communication between processes across node boundaries.

The single image view of the resulting distributed operating system is achieved by a connection layer on top of each LOS. The connection layer's task is to provide network-wide, location-transparent access to all resources of the distributed system. This requires some cooperation between all connection layers while performing a user's request. Typically, the connection layer will provide the function set F' to its users thereby differing from the functionality of the original LOS. However, if the services of the LOS have been defined at an adequate level of abstraction it is conceivable to provide exactly the same functionality F by the connection layer as offered at the LOS-interface. This is the most desirable case since it provides for full portability of programs



LOS	Local Operating System
F	Function Set of LOS
IPC	Inter-Process Communication
T	Function Set of IPC
F'	Function Set of Connection Layer

Figure 8 MIDOS - Architecture of a distributed operating system (Multi-Instance Distributed Operating System)

in the network environment which have been originally developed for running under control of a local operating system.

One of the major problems which had to be solved were the integration of the local file systems in a network-wide file directory scheme[13-15]. Adequate load distribution and load balancing policies and mechanisms are still a matter of ongoing research[16.17].

It is possible to configure a node in a MIDOS-system with zero resources of a given type and removing the corresponding functions from the LOS. This will result in specialized nodes which are dedicated to certain functions. This procedure is equivalent to applying the distribution strategy C as introduced in the previous chapter as a second step to a MIDOS-prestructured operating system. This method has been applied to specialize nodes in a MIDOS-system to discless workstations, file servers etc. The UNIX file servers NFS[18] and AFS[19] are prominent examples for the UNIX world.

For comparison reasons we will use the notion MIDOS-I to denote a system purely structured according to distribution strategy B. The notion MIDOS-II is used in order to denote a system primarily structured along distribution strategy B while the resulting distribution units are subject to the distribution strategy C thereafter.

B. SIDOS-Architecture

The SIDOS-architecture(Single-Instance Distributed Operating System) results by applying distribution strategy C to the original LOS as the primary structuring method. This will lead to a system with specialized servers which manage all resources of a given type and offer a restricted function set for use by the external environment.

In its pure form SIDOS-systems do rarely exist since a single server handling all resources of a given type imposes unnecessary restrictions which is likely to end up in bottlenecks. Therefore distribution strategy B is usually applied as a second step to SIDOS-prestructured operating systems. The resulting distributed systems are usually made up of a certain number of workstations

and specialized servers such as file servers, print servers, network servers etc. The typical architecture of a node in a SIDOS-architecture is shown in Figure 9.Instead of a full function operating system it consists of a simple distributed operating system kernel which provides only the communication infrastructure for the higher layers. The functionality of a node is established by a specific client /server component running on top of the kernel(it is not excluded to run more than one client or server component on a kernel).

For reasons of comparison of the different approaches we will use the notion SIDOS-I in order to denote purely SIDOS-structured systems. The term SIDOS-II will be used in order to denote systems primarily structured according to the distribution strategy C where the resulting structuring units are subject to distribution strategy B thereafter.

A few examples of systems of the SIDOS-class are V[20], ACCENT[21], Amoeba[22], Eden[23], Clouds[24], CONIC[25], DEMOS/MP[26], Cronus[27], INCAS[28] from the research environment. There are also commercial products available which follow the SIDOS structuring approach as for example the Apollo-Domain system.

Figure 10 summarizes the results of the above discussion for further investigations.

The MIDOS-I architecture is unique in the sense that it preserves complete functional autonomy of each node with respect to the function set of the original LOS. If a node fails then all resources managed by this particular node get unaccessible but the remaining nodes can still provide their full service since they do not depend on the availability of other nodes. This property is sometimes referred to as graceful degradation. A disadvantage of the MIDOS-I-architecture is its rather low distribution granularity as compared to the MIDOS-II and SIDOS-II architectures.

Distribution granularity as a measure for the degree of distribution obtained by the application of a certain distribution strategy is a determining factor for such desirable properties as incremental extensibility and support of highly distributed applications. As higher the distribution granularity as smaller are the increments in which a system can grow and as higher is the potential of performing operations in parallel.

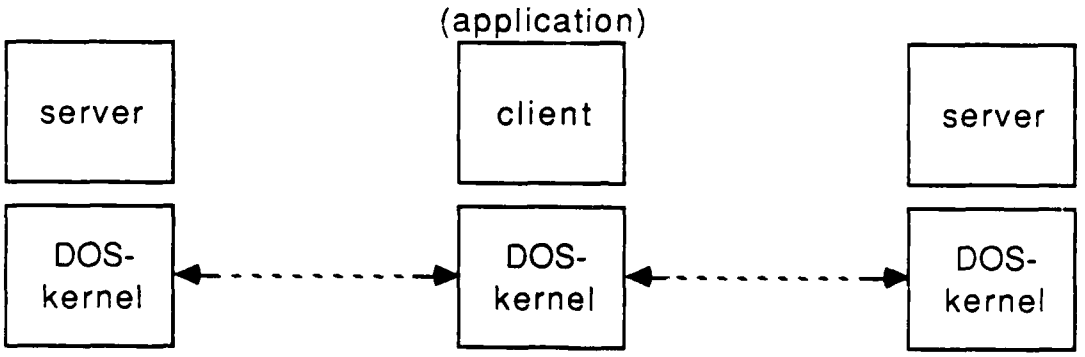


Figure 9 SIDOS - architecture of a distributed operating system (Single Instance Distributed Operating System)

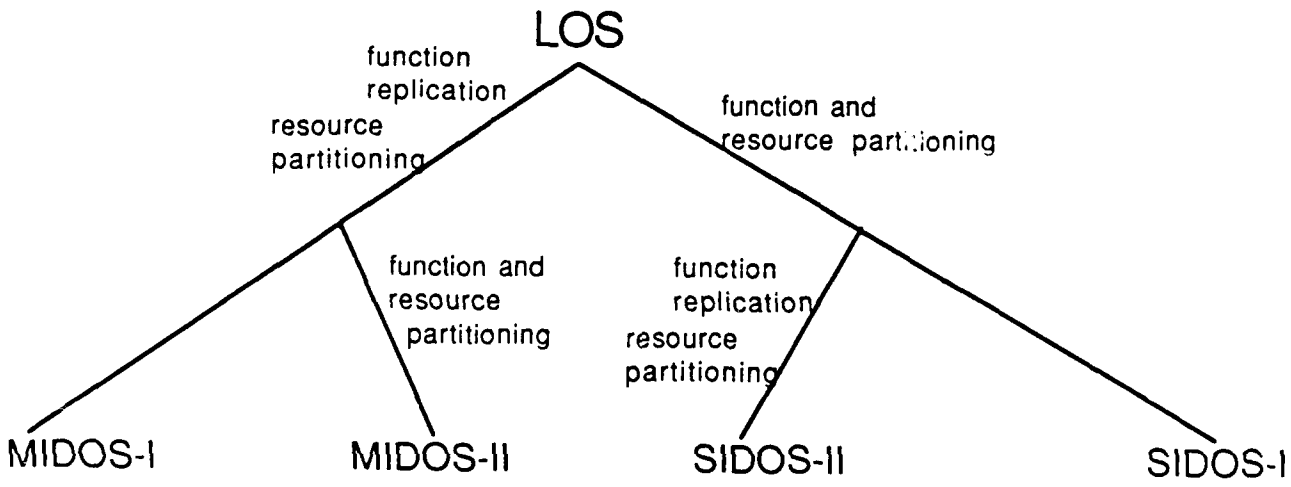


Figure 10 Classification scheme for distributed operating systems

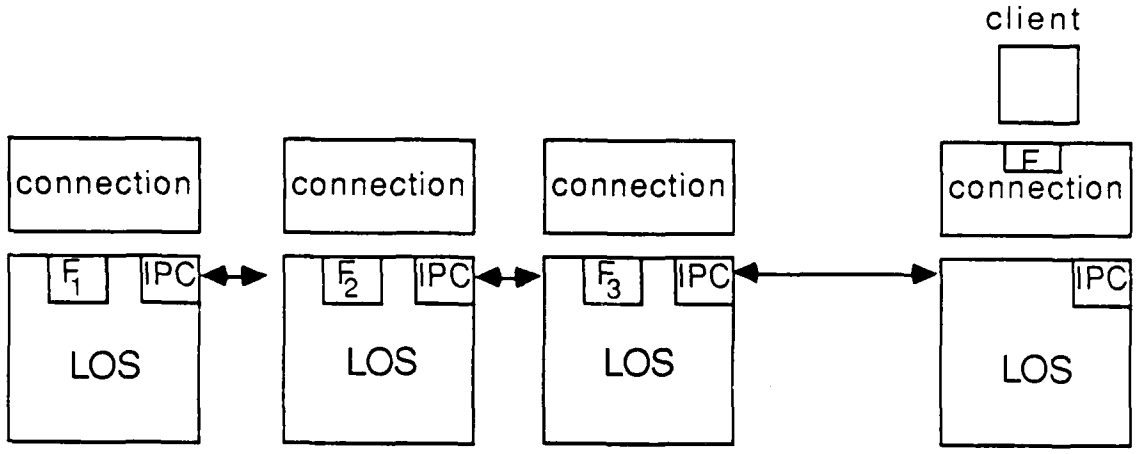
One would expect that the MIDOS-II and the SIDOS-II architectures are equivalent since both exhibit the same distribution granularity(because the same distribution strategies have been applied but in reverse order). For a more objective comparison of the two approaches we will consider an example system consisting of three server nodes and one client node as shown in Figure 11.

A service request issued by a client program in Figure 11a is accepted by the corresponding connection layer first. The connection layer will start a cooperative conversation with all other connection layers until the appropriate location for performing the service has been determined. The corresponding connection layer will then call the requested service from its local LOS and reply to the originating connection layer after completion of the service request. The originating connection layer will finally return control to the blocked client.

Notice that neither a client nor the different LOS's are aware that they are part of a larger system. This global knowledge is completely encapsulated within the connection layers.

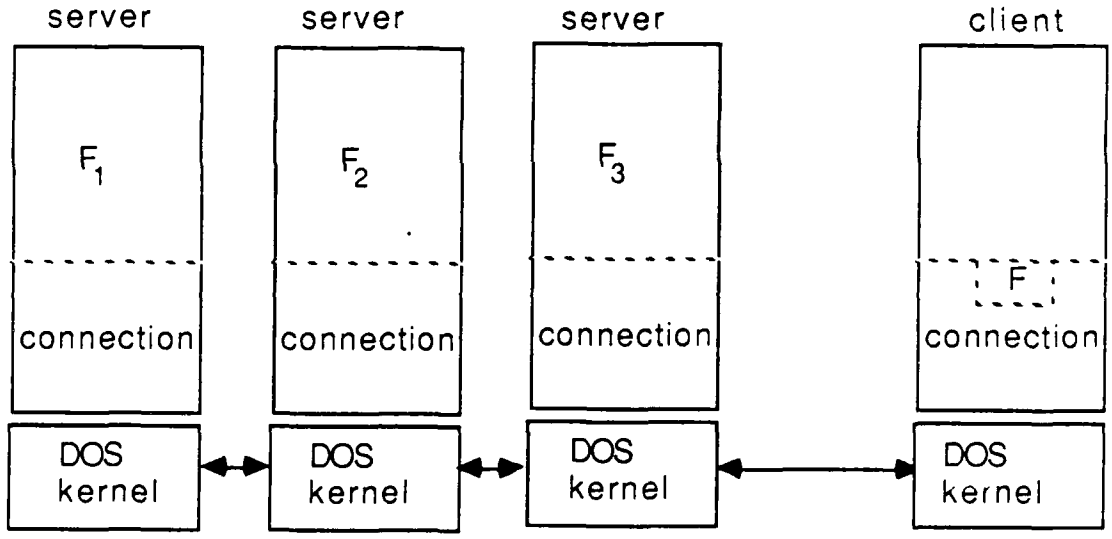
In the SIDOS-II architecture of Figure 11b the *clients view of* the available services of the system is different: it sees a set of servers it can send requests to(being unaware of the server's location).In case that several alternatives are available for requesting a certain service the client negotiates directly with the corresponding servers in order to determine the most suitable one to carry out his request. In order to isolate the global knowledge(how many servers are present) from the local service handling as provided by the MIDOS-approach one can decompose clients and servers into two sublayers each thereby separating these two aspects clearly from each other. This has been indicated by dotted lines in Figure 11b. By doing this the difference between MIDOS-II and SIDOS-II architectures degenerates to a mere software engineering argument:

In the MIDOS-II architecture services are encorporated in the dedicated LOS's. It is required that the overall LOS-structure allows to cut an LOS to any desired functionality necessary for a certain type of service. p In the SIDOS-II architecture services have been clearly separated from lower level infrastructure mechanisms like communication primitives.



$F = (F_1, F_2, F_3)$

a) Example of a MIDOS-II system



b) Example of a SIDOS-II system

Figure 11 An example distributed operating system as a MIDOS-II and SIDOS-II architecture

This yields the following advantages:

-Clients and servers are represented in a uniform way, i.e. there exists only a single abstract concept for their realization.

-The system philosophy is basically open ended with respect to later extensions by new services since the DOS-kernel does not care about the number and functionality of clients and servers running on top of it.

For the above reasons The SIDOS-II architecture is considered superior over MIDOS-II so that we will spend no more time for investigations of the MIDOS-II approach.

Consider now the following thought-experiment: we configure each node of a SIDOS-II structured distributed system by a complete set of servers, one for each different type of services as depicted in Figure 12. Each node can now provide the whole spectrum of OS-functions in an autonomous way as this is the case for MIDOS-I architectures(provided that the required peripheral devices are locally connected) In this way a MIDOS-I system can be viewed as a special case of the SIDOS-II architecture. For these reasons we can abandon further investigations in the MIDOS-I architecture.

Since the SIDOS-I architecture has been ruled out earlier for its limited practical value the SIDOS-II architecture remains the only alternative which deserves a more elaborated consideration in the succeeding chapters.

4. COMMUNICATION MODELS

In this chapter we will focus on the elaboration of basic communication models for distributed operating systems within the framework of the SIDOS-II system class. This requires the development of models for

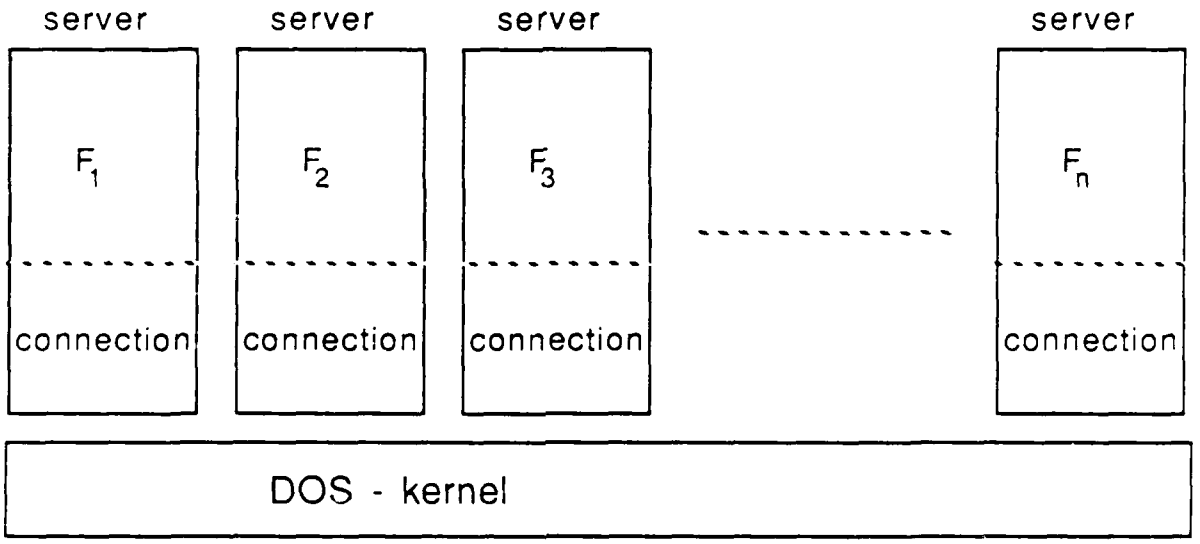


Figure 12 A SIDOS-II structrued operating system providing complete functionality at each node

a. the architecture of interacting units running on top of the DOS-kernel,

b. the interaction scheme imposed to provide controlled communication between interacting units.

A variety of different structuring models have been proposed and implemented in experimental distributed systems.There exist only a few attempts to classify these approaches[29-32]. Our simple classification scheme illustrated by Figure 13 distinguishes between active units represented by processes or teams and passive objects represented by a collection of procedures subject to external invocation.

If units are active they communicate with each other by some form of explicit message passing. Two different variations of message passing scheme have been proposed:

a. One-way message communication

Here the message is merely transported to a given destination by the SEND operation and picked up from the transport system by a RECEIVE operation at the receiver's site later on. The message transaction does not include the execution of some actions at the receiver nor the reply of results. Sending back of results is an independent message transaction issued by the receiver.

b. Two-way message communication

In a two-way communication scheme a message transaction includes

-sending a message to the destination
-accepting the message at receiver's site
-performing an action as a response to the message
-sending back possible results by a reply message.

If units are passive in a two-way communication scheme a remote invocation mechanism is used as the interaction method also known as "Remote Procedure Call"(RPC).The RPC concept is well understood, relatively easy to implement and yields the additional advantage that users are familiar with it[33,34]. The RPC is a two-way communication concept since result parameters can be

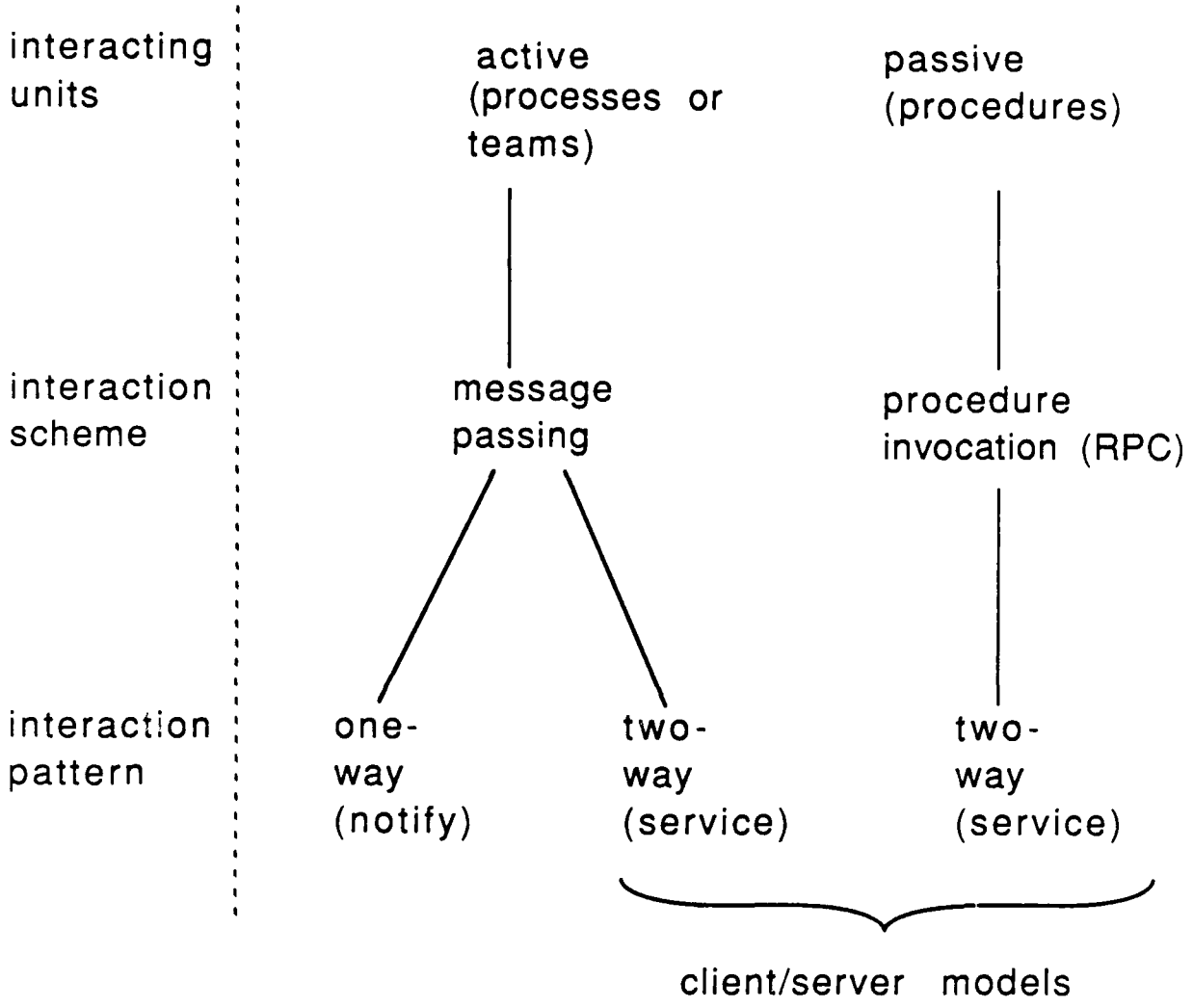


Figure 13 Classification of structuring models

passed back to the caller as with a regular procedure call.

Message passing schemes can be further refined if the degree of synchrony between senders and receivers is taken into account. This leads to the distinction of four basic message passing semantics(Figure 14):

1. Rendezvous semantics

It is a one-way synchronous message passing scheme in which the sender can never send messages faster than received by a receiver, i.e. a rendezvous has to be established between senders and receivers prior to sending the message[35].

2. No-wait semantics

It is a one-way, asynchronous message passing scheme in which senders can issue SEND operations with a speed independent of the corresponding RECEIVE operations of servers 31. This property is achieved by a (principally unlimited)message buffer capacity somewhere in the path between the sender and the receiver. In early-buffering schemes the message will be buffered at the senders site which results in least waiting times within the SEND operation. In late buffering the sender is blocked in the SEND operation until the message has been stored successfully in a message buffer at the receiver's site which increases the waiting time.

3. Remote Service Invocation semantics(RSI)

It is a two-way synchronous message passing scheme which blocks the sender in the SEND operation until a reply message has been properly received from the receiver. RSI mechanisms can serve as a basis for the implementation of the RPC mechanism. Adequately defined they can provide more flexibility than a pure RPC mechanism as demonstrated in the V-kernel[20].

4. Asynchronous Remote Service Invocation semantics(ARSI)

It is a two-way message passing scheme which does not bind the sender at one SEND operation at a time. SEND,RECEIVE and REPLY operations can be issued independently of each other as

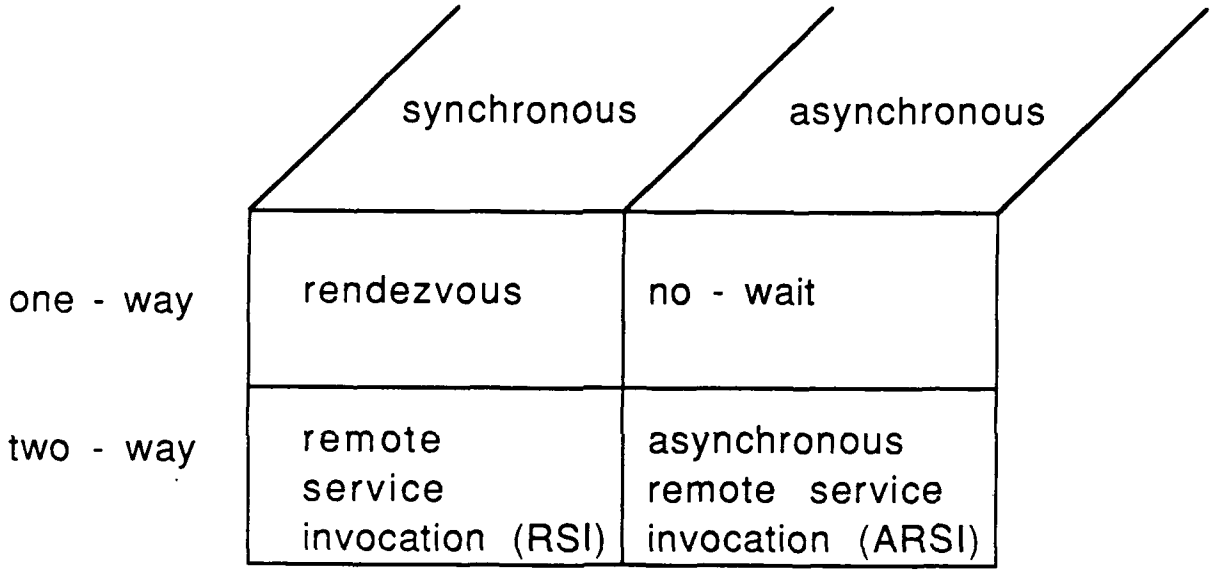


Figure 14 Message Passing Semantics

in asynchronous communication[36]. This scheme is very powerful but also hard to use and difficult to implement.

Another important characteristic of a message passing scheme is the expected degree of reliability. Four different levels of reliability are usually distinguished:

-At-least-once
-At-most-once
-All-or-nothing
-Exactly-once

At-least-once semantics guarantees correct completion of a message transaction at least once. It does not prevent message duplication and should be used only if the operation triggered by the message at the receiver has the property of being idempotent[29].

At-most-once semantics guarantees correct completion of a message transaction at most once. In the absence of permanent node crashes and broken transmission lines it performs the message transaction exactly once.

The all-or-nothing semantics guarantees atomic behavior of a message transaction in spite of crashes of nodes and transmission lines: either it performs its function completely or it has no effect at all[37,38].

Exactly-once-semantics guarantees under all circumstances successful completion of a message transaction in spite of node crashes or broken transmission lines. It can only be approximated to a certain degree by redundancy mechanisms[39].

The two-way communication paradigm is particularly suited if the prevailing form of interactions between users is to request and perform a service. The interacting units play the role of either clients(which request a service) or servers(which provide some sort of service for clients).

Client-server models as represented by the RPC concept or by the two-way message passing

scheme appear as the adequate models for structuring distributed operating systems since their main job is to provide different kinds of services to user programs.

In the succeeding section we will take a deeper look into client server models therefore.

5. CLIENT-SERVER MODELS

In a client-server system the relation between any pair of client-server can be described by the state diagram of Figure 15. In the initial state "service completed" previous services have been completed and new ones not yet started. In the state "service requested" a client has released a service request which has not been taken into account by the server. If the server has accepted the request the state is changed into "service in progress". After completion of the service request the server replies the results back to the client which changes the state back into "service completed".

Figure 16 shows the overall structure of a client-server system. We assume the simultaneous existence of several distributed applications that view the distributed operating system as a collection of servers. Within the distributed operating system there might exist servers not available for general use which are only needed to implement higher level services. In this way the servers of the DOS generally form a hierarchy where only the servers of the bottom layer never turn temporarily into clients. All higher level servers behave as clients against the lower layers and as servers against the upper layers.

It should be noticed that no assumption has been made about the communication model used within distributed applications. While every component of the distributed application requesting a service from the DOS behaves as a client the internal communication model of the distributed application might substantially differ from a client-server model. Implications are discussed in [40].

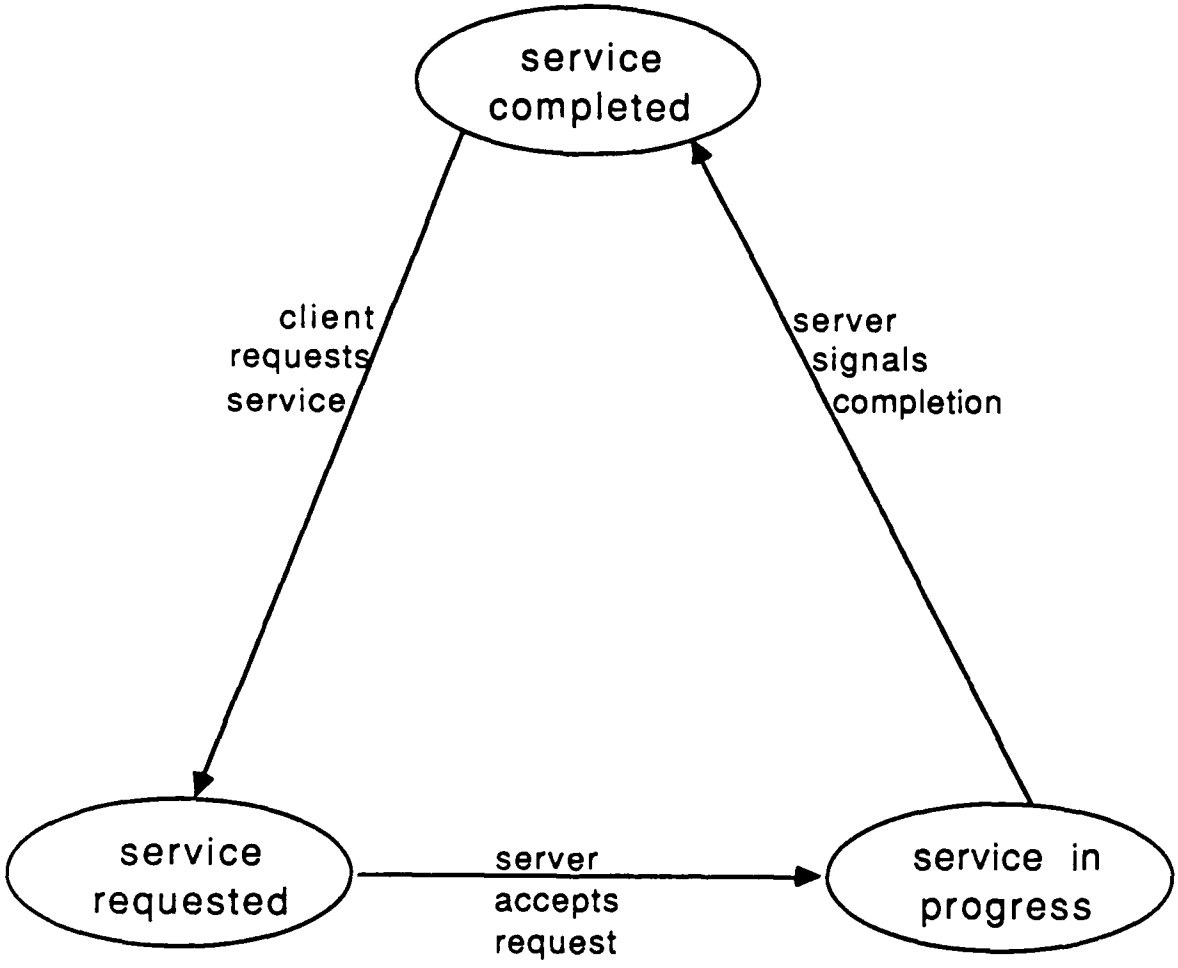


Figure 15 State diagram for Client/Server Systems

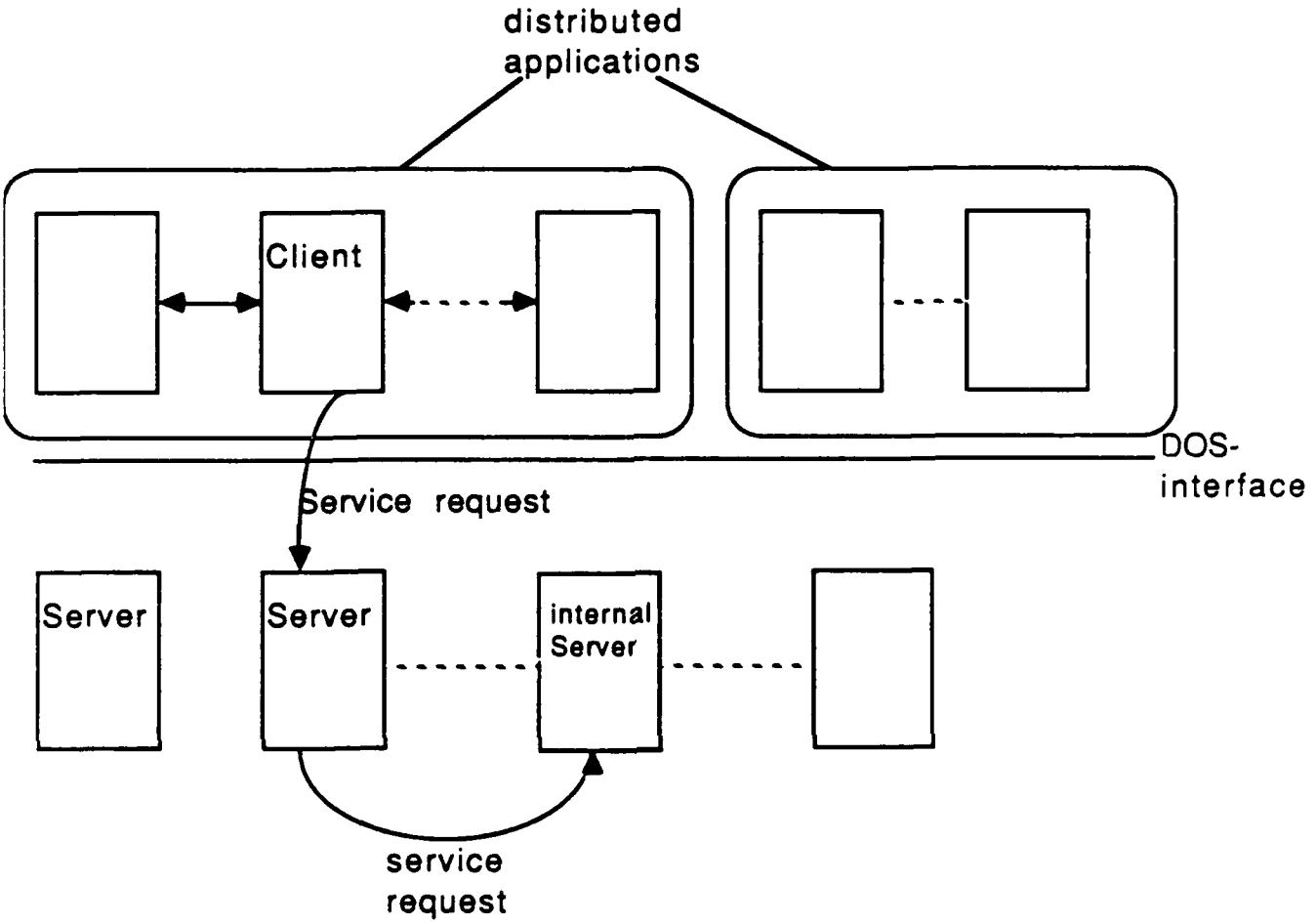


Figure 16 Overall Structure of a Client/Server System

Two different abstract views for client-server systems have been developed[41]:

a) the process/message view

b) the object/action view

We will present both views in detail and discuss the differences (and similarities) as well as possible implications of both approaches.

5.1 THE PROCESS/MESSAGE VIEW

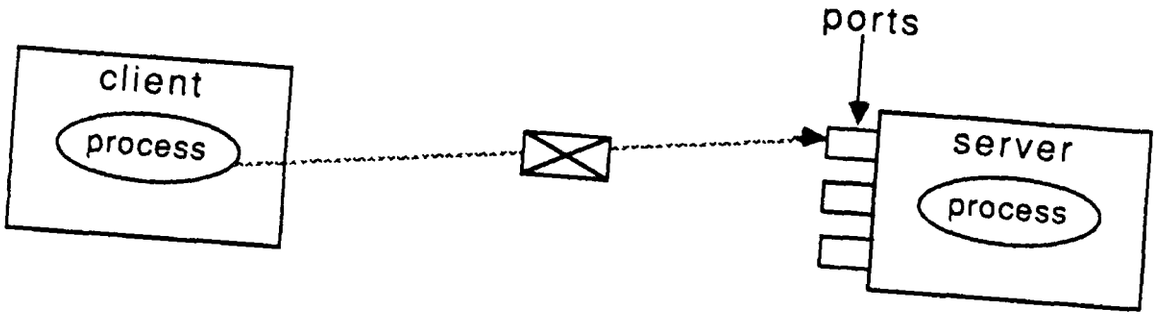
In the process/message view of the client-server model a given system is decomposed into processes or process groups which exchange messages. Figure 17 summarizes the essentials of this view. In its simplest appearance each client-server is represented by a single process as depicted in Figure 17a. Request messages are sent by a client process to a port connected to a server process. Ports represent different service types exported by a server.

A generalization of this view is represented in Figure 17b. Here, each client-server is represented by a team of processes. A request message is issued always by a specific process member of a team. Since the message is directed to a port rather than to a process the internal process structure of a team remains transparent to clients.

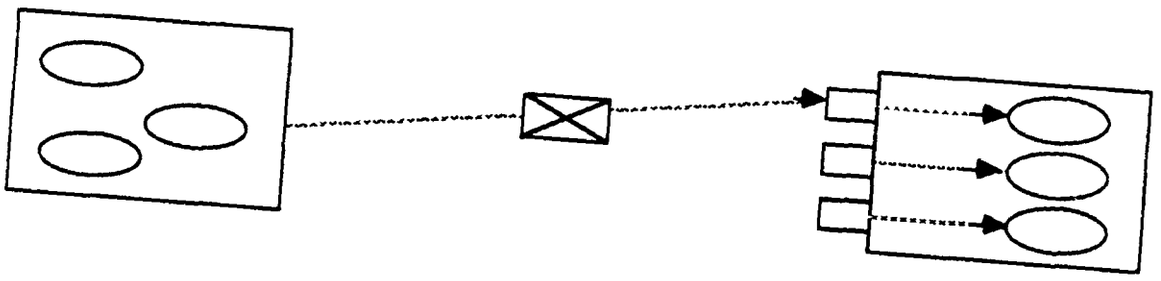
In a static team structure the number of processes forming a team is fixed at team creation time. A usual approach for subdividing the team's work is to associate a process with each port. Simultaneous requests for services arriving at different ports can be processed in parallel conceptually. However, overlapping requests to the same port are queued necessarily.

In a dynamic team structure the number of processes may vary at run time. An obvious approach to organize a server exploiting dynamic process creation/deletion is the following one:

At team creation only one process- the root process- exists. The root process' task is to create a server process with every incoming message:



a) A single process per client/server



b) A team per client/server

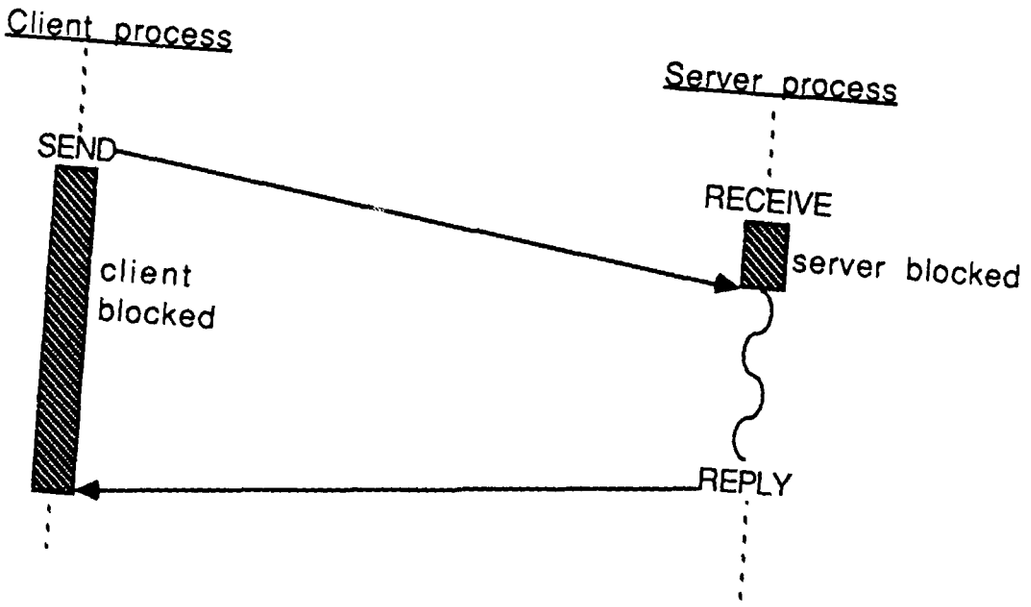


Figure 17 Process/Message View of a Client-Server System

```
PROCESS root
LOOP DO
    Wait for incoming message;
    Create server process;
    Pass PortId to it;
    END;
END root;
```

Depending on the speed of incoming messages an arbitrary number of server processes may come into existence within a team. All server processes adhere to the same basic structure:

```
PROCESS server
  Receive message from port(...);
  Call approptiate service procedure;
  Send reply message to requesting client;
END server;
```

In the above implementation structure the number of processes changes frequently. In contrast to a static team structure the danger of unnecessary serialization of service processing diminishes[32]. The overhead associated with frequent process creation/deletion can be kept at an acceptable level by an efficient light weight implementation. The dynamic approach has been adopted for processes in guardians[42].

The time flow diagram in Figure 17c provides a rough understanding of the semantics of the communication primitives SEND, RECEIVE, REPLY considered as the minimal function set to support the process/message client-server paradigm. The SEND operation will block a client process until the successful arrival of a corresponding reply message. The RECEIVE operation blocks a server until a request message has been arrived and passes the message content (or a pointer to it) to the receiver. The REPLY operation is a non-blocking operation which is issued after completion of a request processing by a server. The successful arrival of the reply message at the destination will unblock the client in its SEND operation.

5.2 THE OBJECT/ACTION VIEW

In an object oriented view of client-server systems servers are conceived as a collection of passive procedures which manage an internal state. An action is performed within an object by calling one of its procedures thereby passing a thread of control to the object. The callee of an object is usually another object which has been called before. At the outermost calling level there exist a certain number of processes which stimulate activity in the system by calling object procedures.

In order to avoid dealing with two different types of structuring units- processes and objects- we take a slightly more generalized view of objects. There, objects can contain a certain number of housekeeping processes which are not directly involved in the processing of service requests. This leads to the following principle structure for objects:

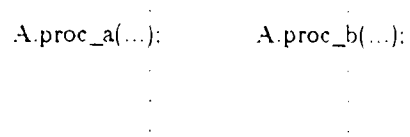
```
OBJECT xyz IS
  ENTRY proc_a(...)- - - - - - - - - - END;
  ENTRY proc_b(...)- - - - - - - - - - -END;
  .
  .
  ENTRY proc_k(...)- - - - - - - - - - END;
data deciarations;
  PROCESS p1 - - - - - - - - - - - - - END;
  .
  .
  .
  PROCESS pn - - - - - - - - - - - - - END;
BEGIN
  Initialization;
END xyz;
```

By a statement of the form

A,B,C: xyz;

we denote the creation of three instances of the object type xyz.

Two concurrently executed call's from within different threads of control of the form



result in two simultaneous actions within the object A. It is assumed that the necessary synchronization between simultaneous actions in an object is supported by appropriate synchronization primitives like semaphores[9] or monitors[10].

Processes can be represented within this generalized object model by objects containing a single internal process each with an empty export interface. An obvious implementation structure for objects as introduced above is shown in Figure 18. At the client's site each Remote Procedure Call(RPC) is redirected to a local stub procedure. This stub procedure collects the actual parameters passed with the procedure call and constructs the message out of it. It then sends the message to a stub process at the receiver's site. The stub process which might have been created in response to the incoming message receives the message and calls an appropriate local service procedure which actually carries out the service request. After return from the service procedure the stub process constructs the reply message, sends it back to the client and terminates.

For convenience reasons it was assumed in Figure 18 that the stub procedure at the client's site and the stub process at the server's site communicate via an synchronous, two-way message handling scheme with the functions SEND, RECEIVE, and REPLY as introduced in section 5.1. This also simplifies the comparison of both approaches.

It is interesting to notice that the implementation structure of the object model sketched by Figure 18 corresponds directly to the server organization with a single root process and a dynamic number of server processes in the process/message paradigm.

This leads to the following primary conclusions:

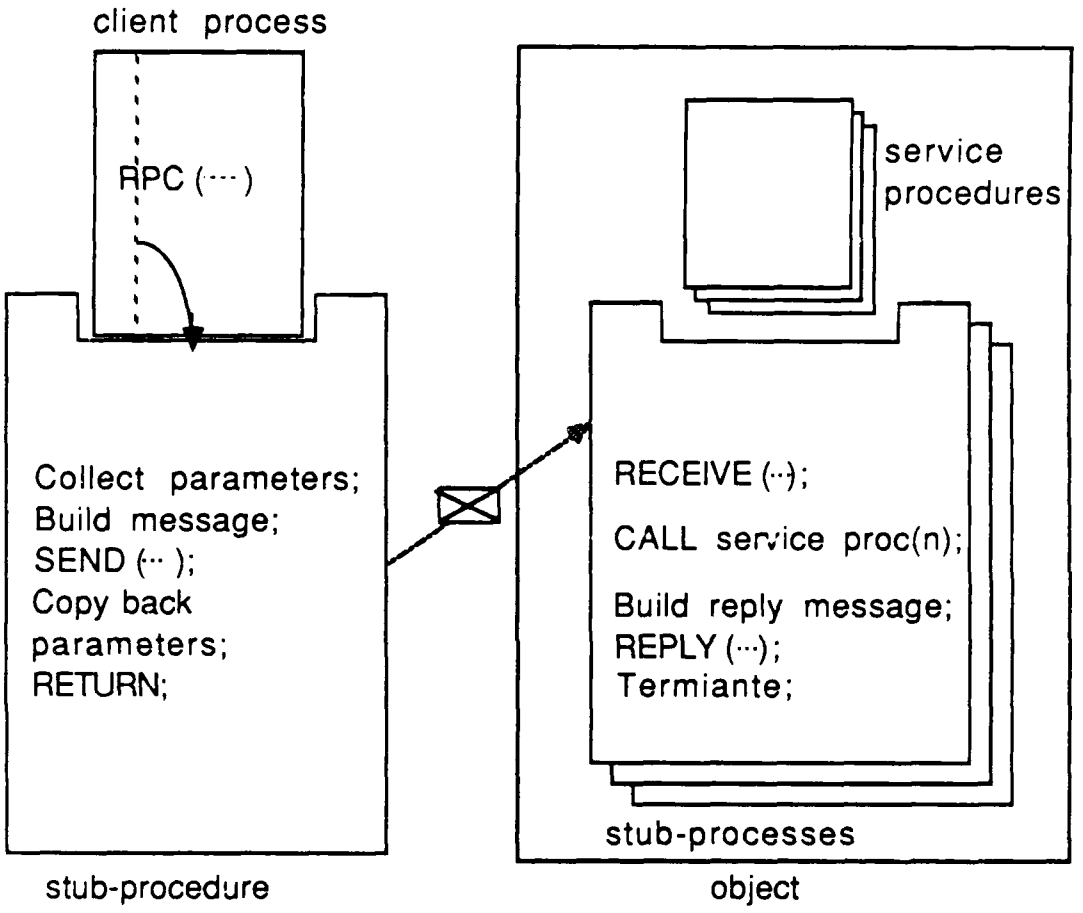


Figure 18 Implementation Structure for the object model

1. At a first glance, the object/action and process/message approach are two abstract views on client-server models with basically the identical expressive power.

2. The object/action view is on a more abstract level. A process/message scheme can easily be used to implement the object/action view.

6 EXTENSIONS TO THE BASIC CLIENT-SERVER MODEL

The basic client-server model as discussed in the previous chapter is sufficient to analyze its principal merits as compared to other approaches. As a basis for a real system, however, the model needs to be extended in several directions in order to serve as a powerful structuring means for distributed operating systems. In the following two subsections we discuss useful extensions in three areas:

1. extensions of the basic client-server model to enhance its expressive power,
2. naming issues,
3. protection.

Since naming and protection are highly interrelated we will treat them together.

6.1 EXTENSIONS OF THE BASIC CLIENT-SERVER MODEL TO ENHANCE ITS EXPRESSIVE POWER

By expressive power we denote the ease of describing typical structures using the available features of a given client-server model. As long as the service to be designed is represented by exactly one single server, the basic client-server models sufficient to model the server as well as the interactions of the server with its environment.

However, the following list of possible interrelationships between a service and servers show other

frequent constellations:

- service = single server
- service = one arbitrary server out of a pool of identical servers
- service = pipelined server formed by n pipeline segments
- service = fault tolerant server formed by m stand-by servers.

Figure 19 provides an example for each mentioned service-server interrelationship. A powerful client-server model should support structuring a service as a pool of identical servers, a pipelined server or a fault tolerant server by convenient addressing and message propagation mechanisms.

An obvious requirement from the above discussion is a selective message broadcast mechanism also called a multicast. An ideal multicast mechanism should reveal the following properties:

a) it should be possible to group targets for messages as for example ports and associate a group name to them.

b) A message sent to a group potentially reaches all actual members of the group.

c) The effort spent by the message handling system to assure a certain degree of success of a multicast operation should be parameter controlled(for example: "enforce successful completion of at least three message deliveries").

d) There is no global control of the members belonging to a multicast group. This releases a sender from specifying the number of potential receivers of a multicast message and favors distributed control algorithms.

Cheriton[20] has shown in the V-kernel design that it is rather simple to extend the basic client-server model to support a flexible multicast mechanisms. Cheriton envisaged the concept of process groups and provided functions for entering and leaving groups. The minimal degree of success of a multicast message transaction is defined as the successful reply of at least a single reply message. However, further expected replies can be received by the GET_REPLY operation.

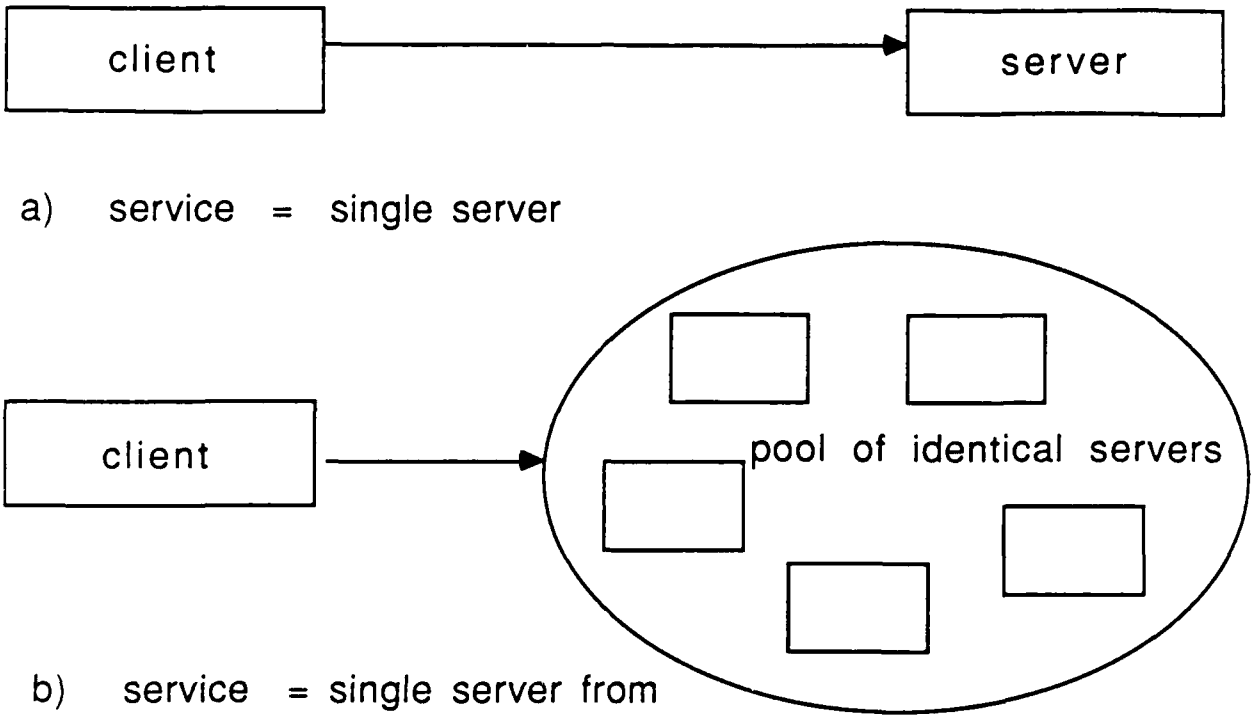


Figure 19 Examples for service-server relationships

There are other extensions to the basic client-server model in the V-kernel which are helpful to structure servers according to needs stated above: The V-kernel offers a FORWARD operation for forwarding a request message from a server who might consider to delegate this job to someone else. This operation has the overall effect that the client-server pair involved in the actual message transaction is redefined replacing the original server by the new one where the request message was forwarded to.

There are further examples taken from the V-kernel which prove to be extremely useful extensions of the basic client-server model with respect to scheduling and efficiency issues:

-The V-kernel allows a server to work simultaneously on several requests (several RECEIVE operations may be issued before the processing of an request has been finished by the REPLY operation). In case of a static process structure this feature regains the server's ability of scheduling incoming requests according to its internal needs.

-The standard message length is 32 bytes. This restriction favors simple storage management policies and fast protocols for the most frequent case. Bulk data may be transmitted in user controlled portions by the COPY_TO and COPY_FROM operations.

The above discussion can be summarized as follows:

In order to support the design of real systems the basic client-server model should be extended to facilitate

- multicast/broadcast
- pipelining (by message forwarding)
- simultaneous request processing by servers
- explicit transfer of bulk data

Prototype systems have proven that it is rather easy to define appropriate extensions in the process/message view and support them efficiently by a DOS-kernel.

It is doubtful that similar extensions could be defined for the object/action view without

destroying its conceptual simplicity. It is our belief that the object/action view as the more abstract model of client-server systems is adequate for certain applications. As such it should maintain a rigid and simple view of objects and actions at the cost of flexibility.

At the DOS level however, more flexibility is needed in order to structure servers and whole systems of clients and servers which requires means for explicit control of message flow, scheduling of arriving messages and data transfer between system components. The process/message view provides the appropriate level of abstraction needed to achieve those requirements.

6.2 NAMING AND PROTECTION

Naming refers to a class of mechanisms in operating systems which help to solve the following problems:

a) generation of unique names
b) name retrieval for public services
c) binding of names to physical locations of the corresponding system objects

In a dynamic system environment new services enter and old services leave the system (to be replaced by improved versions). This requires a mechanism for the generation of unique names even if names exist for a long period of time (several years). If the user starts up a new client it has to get aquanted with those names related to the services it intends to invoke. For example, the client would like to query the system:"give me all names of servers referring to the print service". Names usually consist of rather long character strings. Before an object can be actually referenced the corresponding names have to be bound to physical locations as memory addresses, device addresses etc. In a highly dynamic system environment binding occurs at runtime shortly before the actual reference is made.

In a static system environment the naming problem almost disappears for the following reasons:

- A generation scheme for names is not required since the system designer generates the names and associates them to system components.

- Name retrieval is irrelevant since all names are known and directly used in order to establish the interconnection structure between system components.

-Binding of names to physical locations is carried out at translation or load time.

In a large distributed system environment the incremental growth of the system according to the users needs is highly desirable. Therefore, dynamic system environments are more likely to be the standard case.

Name generation and name retrieval can be organized in a central name server[1]. Central name servers- while sufficient for small distributed systems- can cause substantial performance degradations if they get overloaded in large distributed systems. In addition, the demise of the name server could bring down the whole system.

In a distributed name serving scheme name generation is based on local name generators in each node. Each public server responds to retrieval requests arriving from outside.

In a process/message based system with ports representing services a distributed name retrieval service could be organized as follows: each object offering a public service exports a standard port which is joint in a publicly known port group "PublicServices". A request sent to "PublicServices" will return all port names to the requester currently available for public services.

The knowledge of the name of a server enables a client to contact the server. In the absence of protection mechanisms the ability to contact a service is equivalent to get the service.

Protection mechanisms introduce another fire wall between clients and servers: even if the name of a certain service is known to a client, utilizing the service requires a special permit. If the permit (access right) was not granted before the protection mechanism will reject attempts to request services. In general, protection schemes are responsible for the following tasks[43]:

(a) prevention of unauthorized usage of services
(b) prevention of impersonating of existing services
(c) control of access rights migration

It is still an open question to which degree protection should be supported by operating systems. In trustworthy kernels naming and protection can be included as an integral part in the DOS kernel. Kernels which emphasize protection like HYDRA[44] are often called security kernels. In distributed systems with an open interface like in workstation networks everyone can connect a computer to the network with its personal operating system thereby bypassing any global protection scheme. For this reason some distributed systems don't rely on trustworthy kernels. A good example is the Amoeba system[45].

More research is needed to determine a useful minimal set of services provided by some trustworthy communication interface as the basis of arbitrary protection schemes.

7. THE COOPERATION PRINCIPLE

In a client-server oriented distributed system there exist usually several servers which can carry out a requested service. The decision making process involved in selecting the most suited server(s) requires some form of cooperation between a client and the potential servers before the service can be processed. It is this cooperation process which establishes the fundamental difference between centralized and distributed system organizations (in fact, a centralized system can be viewed as a system with a single server for each service type thus relinquishing cooperation needs).

The objective of any cooperation process is to optimize the overall system behavior with respect to some optimization criteria. Conceivable optimization criteria are:

-averaging resource utilization
-maximizing the overall system throughput
-achieving a required degree of fault tolerance

Our concern in this chapter is to study the principle nature of this cooperation process and to conclude with some hints for future research

It is convenient to think of the cooperation algorithms as being encapsulated in a separate cooperation layer as indicated in Figure 20. This layer is made up of a collection of so called service managers. At client's sites service managers receive service requests from clients. They cooperate with other service managers in order to determine the most appropriate server(s) at that time to carry out the service. The service manager(s) at server's site(s) finally pass a request to the server(s) waiting for incoming requests. Clients and servers are unaware of the decision making process in the cooperation layer underneath.

An ultimate goal of every cooperative effort is to keep the overhead involved in the decision making process small with respect to the amount of information exchanged between service managers and the frequency this information is being exchanged between them.

Suppose in a distributed system exist several identical print servers. A client calling a print operation doesn't care which print server actually performs the request. The corresponding service manager at the client's site starts a cooperation with other service managers in order to determine the print server with the least actual load.

This print server gets finally the request passed to it. An obvious solution for a cooperation algorithm corresponds directly to a well known solution in centralized systems: all service managers at client's sites -after having received a print request- enter a distributed critical region. Within the critical region the present load of all print servers is interrogated and the server with the least load selected. Its present load is adjusted before the critical region is terminated. The print request is finally forwarded to the selected print server. The service manager at a client's site takes the following form:

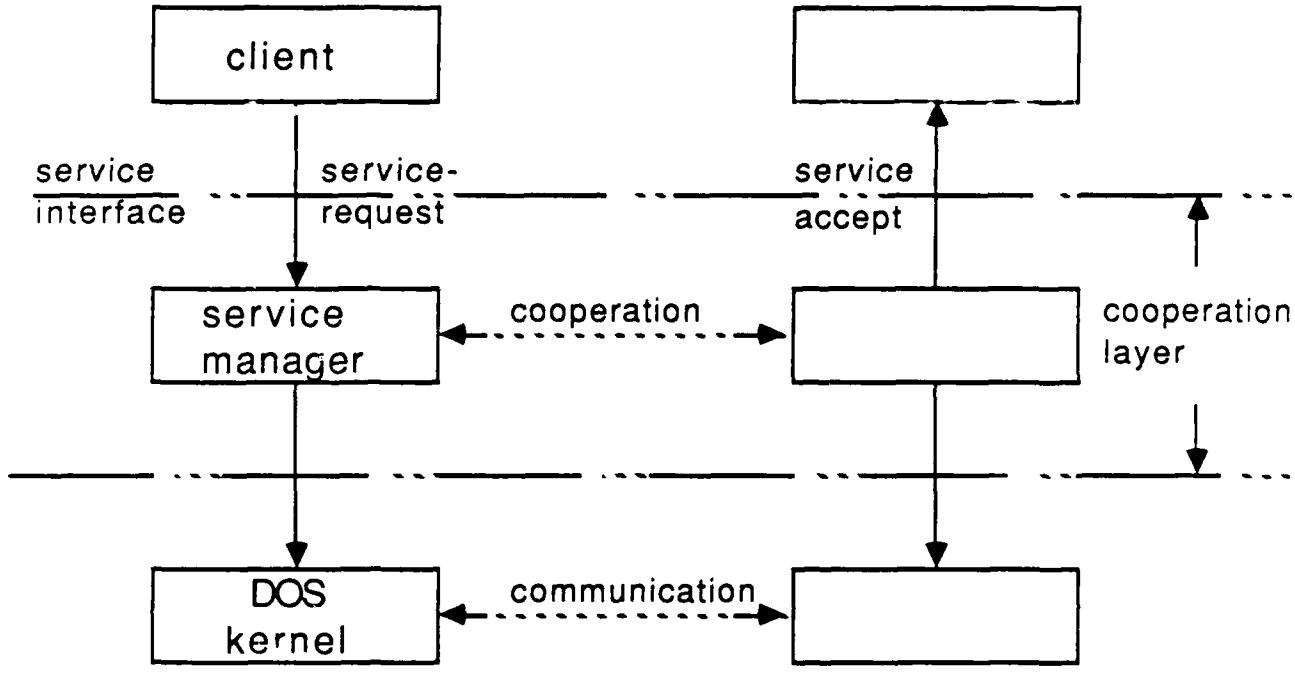


Figure 20 Encapsulation of the descision making process in a cooperation layer

```
ServiceManager (* client site *)
DO
 Receive print request.
 Lock(Mutex); (* distributed P-operation *)
   Broadcast to all service managers at server sites:
   send actual load;
   Collect answers;
   Select server with the least load;
   Adjust load at selected server by
   sending a message to its service manager;
 Unlock(Mutex); (* distributed V-operation *)
 Forward print request to selected server,
END ServiceManager;
```

While the above algorithm works efficiently in a centralized system it might produce prohibitive communication costs in a distributed system due to the expensive distributed critical region[46].

More efficient solutions are based on approximative measures of the actual load of servers, for example by periodic server interrogations. However, they are susceptible to load oscillations.

The above discussion exemplifies a fundamental lack of understanding what factors establish a good cooperation algorithm. It might be an important observation towards a systematic design methodology for distributed algorithms that higher level distributed programs tend to use frequently the same basic algorithms in order to achieve their goal as for example:

- broadcast a message
- obtain a consistent snapshot of a distributed global state
- establish a distributed critical region
- reach consensus on a global situation
- determine global time
- determine termination of a distributed algorithm

etc.[47].

It is considered a challenging research undertaking to devise a design methodology which supports the systematic construction of distributed programs by composition from a universal set of distributed primitives.

REFERENCES

1 A.S Tanenbaum, R.vanRenesse: Distributed Operating Systems, ACM Computing Survey 17, No.4, 419-470 (1985)

2 V.Ambadar: Data Base Machines, Proc. 18th Annual Hawaii International Conf. on System Sciences, 352-372 (1985)

3 R.Bayer, K.Elhardt, W.Kiessling, D.Killar: Verteilte Datenbanksysteme- Eine Uebersicht ueber den heutigen Entwicklungsstand Informatik Spektrum 7, Heft 1, 1-19(1984)

4 S.Ceri G.Pelagatti: Distributed Databases,Principles and Systems, McGraw-Hill 1984

5 T.Haerder,E.Rahm: Mehrrechner-Datenbanksysteme fuer Transaktionssysteme hoher Leistungsfaehigkeit, IT 28, Heft 4, 214-225(1986)

6 D.K.Hsiao(ed.): Advanced Database Machine Architectures, Prentice Hall 1983

7 C.M.Brown, C.S.Ellis, J.A.Feldman, T.J.LeBlanc: Research with the Butterfly Multicomputer. 1984-1985 Computer Science and Computer Engingeering Research Review, Univ. of Rochester 1985

8 K.Frenkel: Evaluating Two Massively Parallel Machines, CACM 29, No. 8, 752-758(1986)

12 P.Enslow: What is a Distributed Data Processing System, IEEE Computer, Vol. 11, No. 1(1978)

13 D.R.Brownbridge, L.F.Marshall, B.Randell: The Newcastle Connection or UNIXes of the World Unite!, Software- Practice and Experience 12, July 1982, 1147-1162(1982)

14 A.Barak, D.Malki, R.Wheeler: AFS,BFS,CFS... or Distributed File Systems for UNIX. Proc. of the 1986 EUUG Fall Conference, Manchester, England, 461-472(1986)

15 M.Kaiserswerth: Verteilte Dateisysteme unter UNIX: Eine Gegenueberstellung verschiedener Loesungen. Informationstechnik it, Heft 6, 390-398(1987)

16 D.Eager, E.Laszowska, J.Zahorjan: Dynamic Load Sharing in Homogeneous Distributed Systems, IEEE Transactions on SE-12, No. 5, 662-675(1986)

17 A.Barak, A.Shiloh: A Distributed Load Balancing Policy for a Multicomputer. Software-Practice and Experience 15, 901-913(1985)

18 NFS Protocol Specification, Networking on the SUN Workstation, SUN Microsystems 1986

19 P.Weinberger: The Eight Edition Remote File System, EUUG Spring Conference, Florence 1986

20 D.Cheriton: The V-Kernel- A Software Base for Distributed Systems. IEEE Software, 19-42(1984)

21 R.F.Rashid, G.G.Robertson: Accent: A Communication Oriented Network Operating System Kernel, Proc. of the 8th SOSP, 64-75(Dec. 1981), also in ACM-OSR 15(5)

[22] S.J.Mullender, A.S.Tanenbaum: The Design of a Capability-Based Distributed Operating System, The Computer Journal, Vol. 29, No. 4, 289-300(1986)

[23] G.T.Almes, A.P.Black, E.D.Lazowska, J.D.Noe: The Eden System: A Technical Review, IEEE Trans. on SE-11, No. 1, 43-59(1985)

[24] P.Dasgupta, R.J.LeBlanc, W.F.Appelbe: The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work, Technical Report GIT-ICS-87/42, Georgia Institute of Technology, Atlanta 1988

[25] J.Kramer, J.Magee, M.Sloman, A.Lister: CONIC- An Integrated Approach to Distributed Computer Control Systems, IEEE Proc. Vol. 130, Part E, No. 1, 1-10(1983)

[26] B.P.Miller, D.L.Presotto, M.L.Powell: DEMOS/MP: The Development of a Distributed Operating System, Software- Practice and Experience 17, 277-290(1987)

[27] R.E.Schantz, R.H.Thomas, G.Bono: The Architecture of the Cronus Distributed Operating System, Proc. of the 6th Int. Conf. on Distributed Computer Systems, IEEE Computer Society, May 1986

[28] J.Nehmer, D.Haban, F.Mattern, D.Wybranietz, H.D.Rombach: Key Concepts of the INCAS Multicomputer Project, Trans. on Software Engineering 13,Vol. 8, 913-923(1987)

[29] E.Jul: A classification of Distributed Operating Systems, Techn. Report 85-05-01, Dept. of Computer Science, Univ. of Washington, 1985

[30] S.M.Shatz: Communication Mechanisms for Programming Distributed Systems, IEEE-Computer, Vol. 17, No. 6, 21-29(1984)

[31] B.Liskov: Primitives for Distributed Computing, Proc of the 7th SOSP, 33-42(1979)

[32] B.Liskov: Limitation of Synchronous Communication with Static Process Structure in Languages for Distributed Computing, Techn. Report No. CMU-CS-85-168, Carnegie-Mellon Univ. 1985

[33] A.D.Birell, J.Nelson: Implementing Remote Procedure Calls, ACM-TOCS 2, 39-59(1984)

[34] A.D.Birell: Secure Communication Using Remote Procedure Calls, ACM-TOCS 3, 1-14(1985)

[35] C.A.R.Hoare: Comunicating Sequential Processes, CACM 21, 666-677(1978)

[36] K.Geihs, R.Staroste, H.Eberle: Operating System Support for Heterogeneous Distributed Systems, GI/NTG-Fachtagung, Aachen Febr. 1987, Informatik- Fachberichte 130, Springer, 178-189(1987)

[37] B.Lampson: Atomic Transactions, Lecture Notes in Computer Science Vol. 105, Springer, 365-370(1980)

[38] J.Walpole, G.S.Blair, D.Hutchinson, J.R. Nicol: Transaction Mechanisms for Distributed Programming Environments, Software Engineering Journal Vol. 2, No. 5, 169-177(1987)

[39] L.Svobodova: Resilient Distributed Computing, IEEE- Software Engineering 10, No. 3, 257-268(1984)

40] J.Nehmer: On the Adequate Support of Communication Interfaces in Distributed Systems, Int. Workshop on "Experiences with Distributed Systems", Univ. of Kaiserslautern, Sept. 1987 (to appear in Lecture Notes in Computer Science, Springer)

41] S.K.Shrivastava, L.V.Mancini, B.Randell: On the Duality of Fault Tolerant System Structures, Int. Workshop on "Experiences with Distributed Systems", Univ. of Kaiserslautern, Sept. 1987 (to appear in Lecture Notes in Computer Science, Springer)

42] B.Liskov, R.Scheifler: Guardians and Actions: Linuistic Support for Robust Distributed Programs, ACM-TOPLAS, Vol. 5, No. 3, 381-404(1983)

43] A.S.Tanenbaum, R.vanRenesse: Reliability Issues in Distributed Operating Systems, Proc. 6th Symposium on Reliability of Distributed Software and Database Systems, Williamsburg, Virginia, 3-11(1987)

44] W.Wulf, E.Cohen, W.Corwin, A.Jones, R.Levin, C.Pierson, F.Pollack: HYDRA: The Kernel of a Multiprocessor Operating System, CACM Vol. 17, June 1974

45] A.S.Tanenbaum, R.vanRenesse, S.J.Mullender: Capability Based Protection in Distributed Operating Systems, Proc. of the Symposium on "Certificering van Software", Utrecht, Netherlands, Nov. 1984

47] G.Roucairol: On the Construction of Distributed Programs, in: Paker, Banatre, Bozyigit(eds): Distributed Operating Systems: Theory Practice, Springer, 47-65(1987)

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION University of Maryland 6b. OFFICE SYMBOL (If applicable) N/A	7a. NAME OF MONITORING ORGANIZATION U.S. Army Strategic Defense Command Office of Naval Research
6c. ADDRESS (City, State, and ZIP Code) Department of Computer Science University of Maryland College Park, MD 20742	7b. ADDRESS (City, State, and ZIP Code) Contr & Acq Mgt Ofc. CSSD-H-CRS, P.O. Box 1500 Huntsville, AL 35807-3801 800 N. Quincy Str. Arlington, Va 22217-5000
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER DASG60-87-C-0066 N00014-87-0241
8c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

11. TITLE (Include Security Classification)
A Structuring Framework for Distributed Operating Systems

12. PERSONAL AUTHOR(S)
Juergen Nehmer

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____		31

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This technical report is an attempt to survey the organisation principles for distributed systems in a systematic and concise manner. Starting with a comprehensive set of terms covering the area of distributed computing, a classification scheme for distributed operating systems is developed. Based on this classification scheme several communication models are surveyed. Client-server models as an attractive structuring means for distributed operating systems are discussed in greater depth. The report concludes by elaborating the nature of cooperation as an unique underlying principle to organise the work in distributed systems.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL