Service Modeling in Distributed Operating Systems

J. Nehmer, F. Mattern

nehmer@informatik.uni-kl.de, mattern@informatik.uni-kl.de

University of Kaiserslautern, Department of Computer Science,
P.O. Box 3049, D-6750 Kaiserslautern, Fed. Rep. of Germany

Abstract

This paper presents a conceptual framework for the organization of services in distributed systems out of collections of participating servers. Our approach is based on the notion of a service layer inserted between the distributed operating system kernels and client-server objects. A service layer consists of disjoint sets of service agents. We demonstrate that this approach favors the development of transparent distributed services which (a) hide the server structure from both clients and servers and (b) encapsulate non-functional issues of a service as, for example, load distribution or fault tolerance within the cooperation protocol established between cooperating service agents. As such, the proposed scheme may be viewed as a generalized framework for modeling arbitrary distributed services as for example a distributed file service.

1. Introduction

Distributed systems represent the most advanced organizational structure of interconnected computer systems. Ideally, the distributed nature of the system architecture is completely hidden from applications. This property is called transparency [ENS78]. Transparency usually comprises name transparency, location transparency, and performance transparency. In distributed systems organized according to the client-server paradigm service transparency is another transparency class of utmost concern. We define service transparency as the invisibility of the server structure involved in a given service. The organization of a service is trivial if it is realized by a single server. However, if several servers contribute to a given service the organization of such services becomes non-trivial because of the distribution of state information.

In the current work, we present a conceptual framework for modeling distributed services out of collections of servers which maintains a high degree of service transparency to both clients and servers. The paper is organized as follows:

Section 2 introduces our basic notion of client-server systems and their architectural constituents. In section 3 various service structures organized by a collection of servers are discussed. In section 4 we introduce the notion of a service layer between the distributed operating system kernel and clients or servers. It consists of disjoint sets of cooperating service agents which coordinate clients and servers in behalf of actual service requests. The examples of services discussed in section 5 demonstrate that the proposed scheme promotes a clean separation between functional and non-functional aspects of a distributed service. Non-functional issues of a service are encapsulated in the cooperation protocol established between the service agents contributing to a given service. In section 6 we briefly mention some open issues.

2. Assumptions on the Basic Distributed Operating System Architecture: The Client-Server Model

Our architectural view of a distributed operating system (DOS) is based on the client-server model and the notion of a DOS-kernel as depicted in Fig. 1 [TAN85]. The DOS-kernels are responsible for the management (e.g. creation and deletion) of client-server objects and the support of location transparent communication between them. Prominent examples for DOS-kernels following the above approach are V [CHE88], Mach [ACC86], Clouds [DAS88], Amoeba [MUL86], Sprite [OUS88], Chorus [ZIM81], and Peace [SCH88]. The existing experimental systems vary in the way client and server objects are defined and in the semantics of the communication mechanisms offered by the kernel. Clients and servers may be single processes or tightly coupled process clusters called teams as provided in V, Mach, and Chorus. In some systems the communication mechanisms merely support primitives for the realization of simple RPC-like communication structures [BIR84] as for example in Peace [SCH88]. More elaborate systems, as for example V and Chorus, offer a broad spectrum of primitives covering asynchronous and synchronous communication as well as point-to-point and multicast communication [SHA84].

The notion of clients and servers as the communicating objects reflects the prevailing request-reply communication pattern assumed between communicating entities. This pattern can typically be observed at the interface between applications and services provided by operating systems. A communication transaction is initiated by a client through a service request which blocks the client until the service has been performed successfully or has failed. Each server contributing to a service waits for incoming service requests. After receipt of a service request the server or a group of servers start processing the request. The servers working on this request might have to exchange parameters and results with the client. This can be done in several ways as, for example, by copy_to and copy_from operations as originally proposed for the V-

0-8186-2088-9/90/0000/0215/$01.00 © 1990 IEEE

kernel [CHE88]. At the completion of a request the participating servers will return a result message. When received by the client, the client deblocks and the request is then considered to be completely processed.

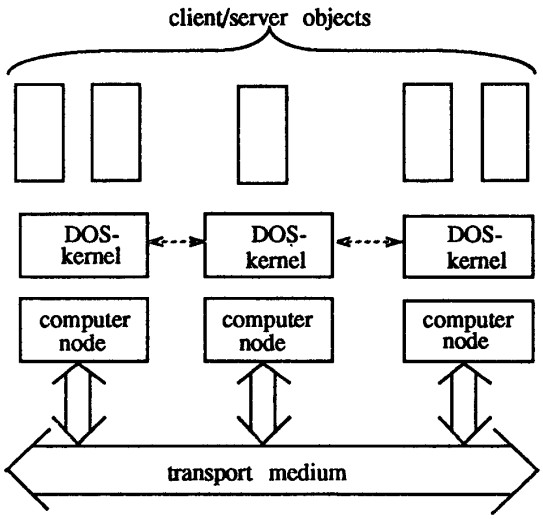

Fig. 1: Basic DOS-architecture

Servers which make use of other servers may temporarily behave as clients. Therefore, the view of a client-server system is dynamic rather than static. At its extreme one can think of a distributed system as consisting of servers only: some of them receive an initial request message in order to get started.

For the purpose of describing various distributed service schemes in the subsequent sections, the following simple communication model is employed which is reminiscent of the actor model originally introduced in [HEW77]:

- Communication is asynchronous and unidirectional.
- Clients and servers are single threaded processes with the following general structure:

```
ACTOR ActorTypeName
Data;

ENTRY ProcedureName (Parameters)
Local variables;
DO
    Statements;
END
:
ENTRY ProcedureName (Parameters)
Local variables;
DO
    Statements;
END

BEGIN
    Initialization;
END ActorTypeName
```

We denote several instantiations of the same actor type by

A, B, C : ActorTypeName

without excluding additional mechanisms for dynamic creation of actors. An actor's operation is activated asynchronously by issuing

ActorName.ProcedureName (Parameters);

It is assumed that the sender of this message is blocked until successful message delivery to the communication subsystem. Message delay is assumed to be undefined but finite. Multicast messages can be sent to all actors of the same type by issuing

ActorTypeName.ProcedureName (Parameters);

Each actor can individually mask specific entries from temporarily accepting messages by the operation

mask ProcedureName;

Arriving messages on a masked entry are buffered until

unmask ProcedureName;

is issued.

Notice that there do not exist operations which imply waiting for an undefined time period as, for example, a receive operation. This allows to schedule actor procedures as atomic actions: an activated procedure runs always through completion before another activation of a procedure within the same actor may occur.

3. The Nature of Services

In this section we study typical client-server interrelationships in the organization of common operating system services. Our investigations are based on the general structure depicted in Fig. 2. It shows a collection of servers which contribute to a given service. The servers are coordinated by a service manager which hides details of the server structure to clients and other servers. Clients access the service always through the service manager. They are unaware of the underlying server structure. The symmetric service transparency implied with this organization has the following advantages:

1. The client's code remains independent of the number of servers involved.
2. The server's code remains independent of the number of other participating servers.
3. Specific coordination efforts which aim at better load distribution, fault tolerance etc. of a service are encapsulated within the service manager. Ideally, the coordination algorithms are not part of the client's and server's code. As such, clients and servers are unaware of non-functional aspects of the service they are contributing to. This again has the advantage that changes to the quality of a service (e.g., degree of fault tolerance) does not affect the clients and servers involved.

The service manager can be modeled as an actor which offers the entries 'ServiceRequest', 'ReadyForService' and 'Result' to clients and servers. 'ServiceRequest' may be viewed as a template which is replaced by a number of actual functions associated to a specific service. The general structure of a service manager may be described as follows:

```
ACTOR ServiceManagerType
    Server[1],..., Server[N]: Server;
    Ready: array[1..N] of bool;

ENTRY ServiceRequest ( Parameters, ResultEntry ):
DO
    IF no server ready THEN queue the request;
        ELSE
            Determine the servers;
            Server[i].Request (...);
            Ready[i] := false;
    END
END

ENTRY ReadyForService ( Server_ID )
DO
    IF request pending THEN Server_ID.Request(...);
                       ELSE Ready[Server_ID] := true;
    END
END

ENTRY Result ( Parameters )
DO
    IF service completed
        THEN
            Determine originating client;
            Client.Result (...);
        ELSE store the intermediate result;
    END
END
:
END ServiceManagerType
```

A client entering a service by ServiceManager.ServiceRequest (Parameters, ResultEntry) has to provide an entry 'Result' for the subsequent delivery of the results by the service manager.

Each server offers as many 'Request' entries as there are different functions defined for this type of service. The general structure of a server is shown below:

```
ACTOR ServerType
State information;
:
ENTRY Request ( ... )
DO
    Process the request;
    ServiceManager.Result (...);
    Decide whether ready for servicing further requests;
    ServiceManager.ReadyForService (own_ID);
END
:
END ServerType
```

There exist at least four typical situations we are faced with in the organization of operating system services:

1. Services are established by exactly one server each.
2. Services are established by a pool of identical servers (e.g., a printer pool).
3. Services are organized by a group of cooperating experts (e.g., a server pipeline).
4. Fault tolerant services using server replication.

Before we proceed with the discussion of specific examples for the organization of distributed services we have to refine the logical structure of a service organization as depicted by Fig. 2.

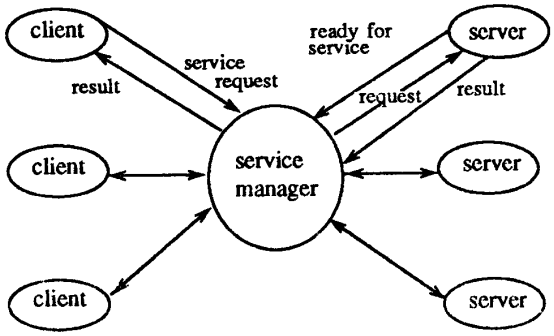

Fig. 2: A general service model

4. Service Agents: A Conceptual Approach to Organize Distributed Services

In a truly distributed system one strives to avoid centralized control structures in order to retain the potential benefits of distribution. The logical structure of a service organization as shown in Fig. 2 can be mapped into two alternative distributed structures if we distribute the concept of a service manager in an adequate way. In both cases, the functionality of the service manager is realized by a set of cooperating service agents.

Fig. 3 shows the first distributed solution called a client-oriented distributed service organization. It is obtained from Fig. 2 by distributing the functionality of the service manager of Fig. 1 to service agents associated to clients (for ease of presentation we assume a one-to-one correspondence of clients to service agents although efficiency arguments might lead to other constellations).

The general client-oriented service processing paradigm may be described as follows:

1. A client requests a service from its associated service agent.
2. The service agents maintain local state information on the servers' states.
3. After receipt of a request a service agent negotiates with all other service agents in order to achieve agreement on the servers to get involved in the service.
4. The service agent forwards the request directly to the selected servers.
5. The servers reply to the service agent after processing of the request.
6. The service agent returns the result to the client.

The concept of distributed critical regions as proposed originally by Lamport in [LAM78] is an example for a client-oriented distributed service. Service agents provide P and V operations on distributed semaphores to their clients. They maintain local state information on each semaphore in the system.

Global cooperation between service agents takes place on each call of a P and V operation with the intention to keep the local views on the semaphore states consistent. This approach can be employed in order to synchronize the access to servers if a distributed semaphore is associated with each server.

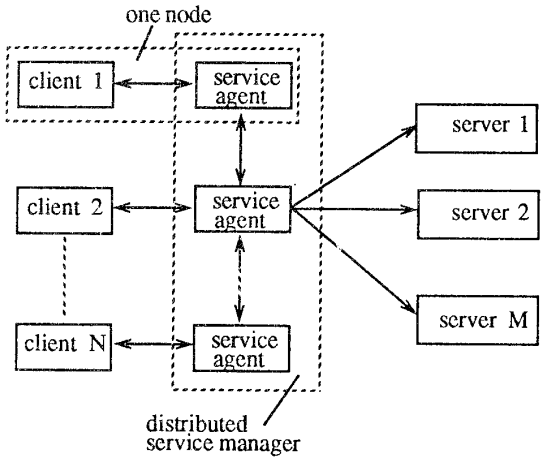

Fig. 3: Client-oriented organization of a distributed service

Alternatively, one can distribute the functionality of the service manager of Fig. 1 to a set of service agents associated in one-to-one correspondence to servers. The resulting structure is depicted in Fig. 4.

The general server-oriented service processing paradigm for this organization scheme may be described as follows:

1. Each server signals its readiness to work on a new service request to its associated service agent.
2. A client requests a service by issuing a multicast service request to all service agents of a given service (or by sending the request to an arbitrarily selected service agent).
3. The service agents cooperate with each other in order to determine the server or the group of servers which are suited best to handle the request.
4. After processing the request the server communicates the results to its corresponding service agent.
5. A predetermined representative of the set of service agents finally returns the result to the initiating client.

An example of this server-oriented distributed service organization is the management of a pool of identical resources as, for example, printers: A client multicasts its request to the set of service agents each responsible for managing a particular print server. The service agents determine - by cooperation - an idle print server (or wait until a print server becomes idle). Print orders from the client are then forwarded by the corresponding service agent to the selected print server until the printer is released.

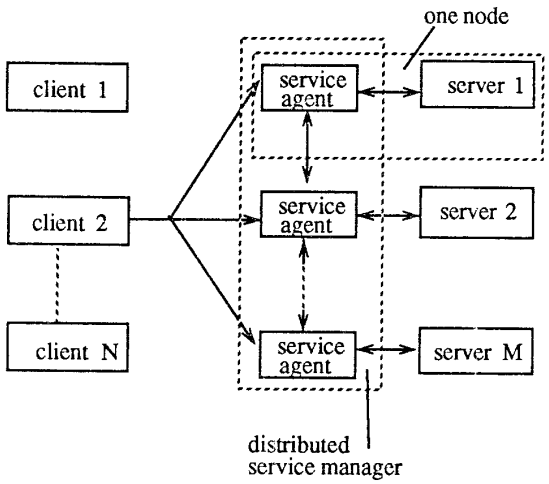

Fig. 4: Server-oriented organization of a distributed service

Besides the two extreme organizations for distributed services sketched above there might be good reasons for a hybrid approach which combines the advantages of a client-oriented and server-oriented organization for distributed services: the service manager of Fig. 2 is distributed into a set of service agents for clients (SACs) and a set of service agents for servers (SASs). This approach is illustrated in Fig. 5. The service agents can be thought of constituting a separate service layer between the client-server layer and the operating system kernels. Ideally, this symmetric approach has the advantage that changes in the service approach (i.e., client-oriented versus server-oriented) remain completely transparent to the clients and servers themselves.

Designed as independent actors, the service agents follow the structure shown below:

```
ACTOR SAC    /* service agent for clients */
State information;
:
ENTRY ServiceRequest ( Parameters )
DO
    Negotiation with other service agents;
    Distribution of the service request to a
    subset of other service agents (SASs);
END
:
ENTRY Replies ( Result )
DO
    collect the results;
    client.Result (Result); /* return result to client */
END
:
END SAC
```

```
ACTOR SAS       /* service agent for servers */
State information;
:
ENTRY ReadyForService ( Server_ID )
DO
    Ready[Server_ID] := true;
END
:
ENTRY Result ( ... )
DO
    /* Send result to responsible SAC */
    SAC[i].Replies (...);
END
:
END SAS
```

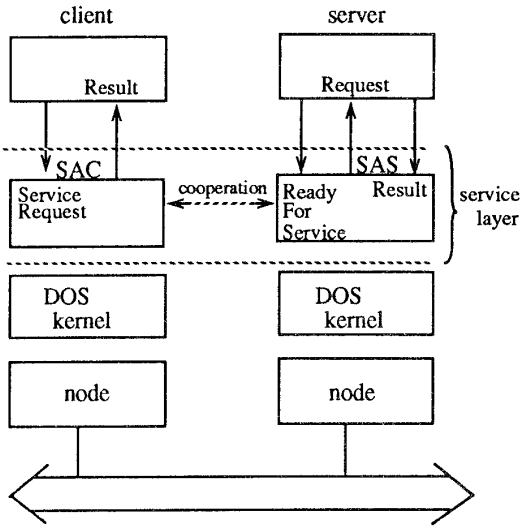

Fig. 5: Distributed system architecture with a service layer

5. Examples

In this section we present two examples which demonstrate that the non-functional aspects of a distributed service can be encapsulated in a separate service layer. This service layer is formed by cooperating service agents; to a large degree its structure is transparent to the clients and servers. Due to space restrictions the examples are somewhat simplistic.

Example 1: A Pool of Compute Servers

Assume that a pool of equivalent compute servers exists. The task of a compute server is to execute one or several processes on a computing node and to manage the local resources (cpu, memory, input-output links). When a client process wants to create a new process it calls the service "determine an appropriate compute server". In a simplified model "appropriate" could just mean the node with minimal load. A reasonable implementation would use the server-oriented service organization paradigm. Each compute server has a corresponding service agent which is kept informed about the load of its compute server. Whenever a service agent receives a request to determine a compute server with a minimal load, it starts an inquiry round. This could be realized, for example, by some broadcast mechanism using an echo algorithm covering all service agents [CHA82]. For ease of presentation a virtual ring going through all service agents is assumed in the example below.

```
ACTOR SAS_CompServer
    NextSAS: SAS_CompServer;
    CurrentLoad: float;

ENTRY NewLoad (load)  /* called by the server */
DO
    CurrentLoad := load;
END

ENTRY AcceptRequest
    DO
        Store identity of the service caller;
        NextSAS.Forward (CurrentLoad, ID, ID)
        /* ID is the identification of the current actor */
    END

ENTRY Forward (load, minID, origin)
DO
    IF origin = ID
    THEN
        return minID to service caller;
    ELSE
        IF CurrentLoad < load
        THEN
          NextSAS.Forward (CurrentLoad, ID, origin);
        ELSE
          NextSAS.Forward (load, minId, origin);
        END
    END
END

END SAS_CompServer
```

After a full round of the control message the originator knows the ID of the node with minimal load. It should be noted that this yields only an approximation since the situation might have changed while the inquiry message was circulating. Another drawback of this simple solution is that several clients initiating the service independently from each other might get the same result thereby overloading the compute server. These problems can be solved by using more intelligent heuristics and cooperation strategies (e.g., inquiring only a subset of compute servers, propagating the inquiry message in parallel to the service agents, making a random choice among several appropriate computer servers) involving service agents for servers as well as service agents for clients. We would like to emphasize, however, that these cooperation algorithms are completely located within the service layer, they are neither visible to the servers nor to the clients.

Example 2: A Fault Tolerant Service

In this example we demonstrate the transparent realization of a fault tolerant service. The service provides fault tolerant access to a virtual single resource organized by server replication and the well known principle of hot-stand-by redundancy. The general structure of the service organization is depicted in Fig. 6. Here, the service is organized by two identical copies of the server. It is accessed by three functions: acquire, use, and release. In order to simplify matters it is assumed that the following conditions hold:

- Only fail-stop failures for servers are considered.
- The probability of having more than one server failed at the same time is negligible.
- The servers are autonomous, i.e. they don't involve other servers in order to provide their intended services.
- Messages never get lost by the communication medium.

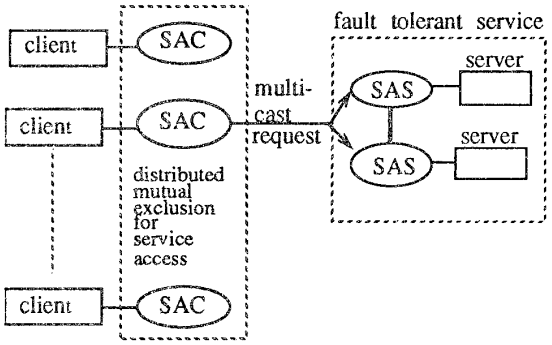

Fig. 6: Organization of a fault tolerant service

With respect to the general service organizations as outlined in section 4 a hybrid approach is used: a client-oriented organization is assumed for 'acquire' and 'release', while 'use' is organized by a server oriented approach. In the skeleton algorithms below we don't show the distributed mutual exclusion scheme [SAN87] for 'acquire' and 'release' used by the SACs. A client issues a 'use' request only when it is granted exclusive access to the service.

The cooperation protocol between service agents for 'use' follows the following simple pattern:

- Each 'use' - request is multicast by the responsible SAC to the corresponding SASs.
- 'Use' requests are queued in both SASs.
- Each client sequentially numbers the 'use' requests.
- Both server replicas eventually process the request and reply their result to their SASs which - in turn - return the results to the originating SAC.
- Result replicas - if received by a SAC - are discarded. The first result is forwarded to the client issuing the 'use'-request.

The simple protocol defined above guarantees continued service even in case of a (single) server breakdown. The crucial part of the protocol is the reintegration of a previously crashed and now restarted server: it must be assured that the next request message accepted for processing by the server is consistent with the internal state resumed after restart of the server. This condition is satisfied by the following protocol:

1. The SAS of a restarted server requests the current state S of its stand-by server (via the corresponding SAS) and the queue of unprocessed messages stored in the stand-by SAS.
2. After arrival of the server state S and its associated queue the SAS of the restarted server checks whether it has request messages in its local queue which have a higher sequence number than those contained in the queue received. Such request messages are kept in the local queue, all other messages in the local queue are remove. (An SAS records the highest sequence number per client which has been processed. Messages which are in the local queue with a higher timestamp than messages in the received queue have not yet been received by the remote SAS when the queue was sent.)
3. The requests contained in the received queue are inserted into the local queue.
4. If the local queue is not empty, the first request is sent to the server.

The resulting structure of the service agents is sketched below.

```
ACTOR SAC
:
:   acquire, release not shown here
:
ENTRY use ( Parameters ) /* from client */
DO
    /* Multicast the request to both SASs */
    SAS.ServiceRequest ( Parameters );
END

ENTRY Replies ( Result ) /* from SAS */
DO
    IF first reply THEN client.Result (result);
            /* return result to client */
END
END SAC


ACTOR SAS
    RemoteSAS: SAS;
    RestartPhase, ServerReady: bool;
    Queue: array [1..M] of ...;

ENTRY ReadyForService () /* from server */
DO
    IF   RestartPhase
         THEN
           RemoteSAS.StatusRequest ();
         ELSE
         IF pending request
              THEN
                  server.use ( ... );
                  ServerReady := false;
              ELSE
                  ServerReady := true;
             END
    END
END
```

```
ENTRY Result ( ... ) /* from server */
DO
    Send result to responsible SAC;
END

ENTRY StatusRequest () /* from remote SAS */
DO
    Server.StatusRequest;
END

ENTRY StatusFromServer (S)
DO
    RemoteSAS.Status (S, Queue)
END

ENTRY Status (S, RemoteQueue)
DO
    FOR each M in Queue
    DO
        IF sequence_no(M) < highest sequence_no of
           corresponding client in RemoteQueue
           THEN
               remove M from Queue;
        END
        Queue := Queue + RemoteQueue;
        RestartPhase := false;
        IF #Queue > 0 THEN
               server.use (...)
               ServerReady := false;
        END
    END
END

ENTRY ServiceRequest ( Parameters ) /* from SAC */
DO
    IF   RestartPhase or not ServerReady
         THEN
           buffer request message in Queue;
         ELSE
           server.use ( ... );
           ServerReady := false;
    END
END
:
END SAS
```

6. Conclusions

In this paper we have presented some preliminary ideas for the organization of services in a distributed system by means of a separate service layer. The service layer is realized by a set of cooperating service agents which provide service transparency to clients as well as to servers. Although the proposed scheme seems to be promising with respect to the stated goals we are still far away from a well understood methodology. Some interesting questions are for example:

- Is it possible to develop a few but universal cooperation protocols for certain service classes?
- To what extend can the cooperation schemes be made generic (e.g., regarding actual system configurations).
- How scalable are the involved algorithms?
- What are useful and required communication and synchronization concepts (e.g., atomic actions, RPC, interrupts)?
- How much knowledge do clients and servers (resp. SACs and SACs) need to have about each other in order to do their work efficiently? (That is, what is the degree of transparency that can be achieved?)

These and related questions and problems concerning the structured organization of service layers will be addressed in the future.

References

ACC86 M. Accetta, R. Baron, W. Balasky, D. Golub, R. Rashid, A. Tevanian, M. Young: Mach: A New Kernel Foundation for UNIX Development, Summer USENIX-Conference, June 1986

BIR84 A.D. Birell, J. Nelson: Implementing Remote Procedure Calls, ACM-TOCS, Vol. 2, 39-59 (1984)

CHA82 E.J.H. Chang: Echo Algorithms-Depth Parallel Operations on General Graphs, IEEE Transactions on Software Engineering, Vol. SE-8:4, 391-401 (1982)

CHE88 D.R. Cheriton: The V Distributed System, CACM, Vol. 31, No. 3, 314-333 (1988)

DAS88 P. Dasgupta, R.J. LeBlanc Jr., W.F. Appelbe: The Clouds Distributed Operating System, Proc. 8th ICDCS, June 1988

ENS78 P. Enslow: What is a Distributed Processing System?, IEEE Computer, Vol. 11, No. 1, 13-21 (1978)

HEW77 C. Hewitt: Viewing Control Structures as Patterns of Passing Messages, Artificial Intelligence, Vol. 8, 323-364 (1977)

LAM78 L. Lamport: Time, Clocks, and the Ordering of Events in a Distributed System, CACM, Vol. 21, No. 7, 558-564 (1978)

MUL86 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)

OUS88 J.K. Ousterhout, A.R. Cherenson, F. Douglis, M.N. Nelson, B.B. Welch: The Sprite Network Operating System, Computer, Vol. 21, No. 2, 23-36 (1988).

SAN87 B.A. Sanders: The Information Structure of Distributed Mutual Exclusion Algorithms, ACM-TOCS, Vol. 5, No. 3, 284-299 (1987)

SCH88 W. Schröder: PEACE: A Distributed Operating System for an MIMD Message Passing Architecture, Proc. 3ICS, 302-312 (1988)

SHA84 S.M. Shatz: Communication Mechanisms for Programming Distributed Systems, IEEE Computer, Vol. 17, No. 6, 21-29 (1984).

TAN85 A.S. Tanenbaum, R.v. Renesse: Distributed Operating Systems, ACM Computing Surveys 17, 419-470 (1985)

ZIM81 H. Zimmermann, J.S. Banino, A. Caristan, M. Guillemont, G. Morrisset: Basic Concepts for the Support of Distributed Systems: The CHORUS Approach, Proc. 2nd ICDCS, 60-66 (1981)