Framework for the organization of cooperative services in distributed client-server systems

Jürgen Nehmer and Friedemann Mattern* discuss the notion of server transparency and the structure of a service layer in distributed systems

A framework for the organization of services in distributed operating systems according to the client-server model is presented. Our approach is based on a clear separation of the concepts of services and servers. It is argued that clients need an interface to services rather than to individual servers. A service is a structure which provides clients with a well-defined functionality with selectable quality. Services are realized by a set of collaborating servers. The proposed framework for the service organization is based on the notion that a service layer is established between the operating system kernel and client-server objects. We demonstrate that this approach supports the development of truly transparent distributed services which hide the server topology from both clients and servers, and which encapsulate non-functional properties of services in cooperation protocols located within the service layer.

Keywords: distributed operating system, transparency, client-server architecture, service layer

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*[1]. Transparency usually comprises name transparency, location transparency, and performance transparency. In distributed systems organized according to the client-server paradigm, *server transparency* is another very important transparency class. We define server transparency as the invisibility of the server topology 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 this paper we present a conceptual framework for modelling distributed services out of collections of servers. The proposed architecture maintains a high degree of server transparency to both clients and servers.

The next section introduces our basic notion of client-server systems and their architectural constituents. The standard interaction pattern between clients and servers is discussed, the notion of *service* is introduced, and a model which maps services to cooperating servers is presented. 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 on behalf of actual service requests. The examples of services discussed in the following section demonstrate that the proposed scheme promotes a clean separation between functional and non-functional aspects of distributed

Department of Computer Science, University of Kaiserslautern, PO Box 3049, D-6750 Kaiserslautern, Germany
*Department of Computer Science, University of Saarland, Im Stadtwald 36, D6600 Saarbrucken, Germany

service. We conclude with a brief summary of open research problems.

CLIENT-SERVER MODEL

Our architectural view of a distributed operating system is based on the *client-server model* and the notion of a distributed operating system kernel as depicted in Figure 1[2]. *Clients* are entities which request and use a service (such as file access or directory service), whereas *servers* are the service offering entities. The kernels are responsible for the management (e.g. creation and deletion) of client and server objects and for the support of location transparent communication between them. Prominent examples of distributed operating system kernels that follow this approach are V[3], Mach[4], Clouds[5], Amoeba[6], Sprite[7], Chorus[8], and Peace[9]. 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[10]. More elaborate systems such as Amoeba, V and Chorus offer a broad spectrum of primitives covering asynchronous and synchronous communication as well as point-to-point multicast communication[11].

The notion of clients and servers as the communicating objects reflects the prevailing *request-reply communication pattern* between communicating entities which can typically be observed at the interface between applications and services provided by operating systems. In this pattern, 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. A server working on this request might have to exchange parameters and results with the client and other servers. This can be done in several ways as, by *copy_to* and *copy_from* operations as originally proposed for the V-kernel[3]. At the completion of a request, the participating servers will return a result message. When received by the client, the client unblocks and the request is then considered to be completely processed. 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.

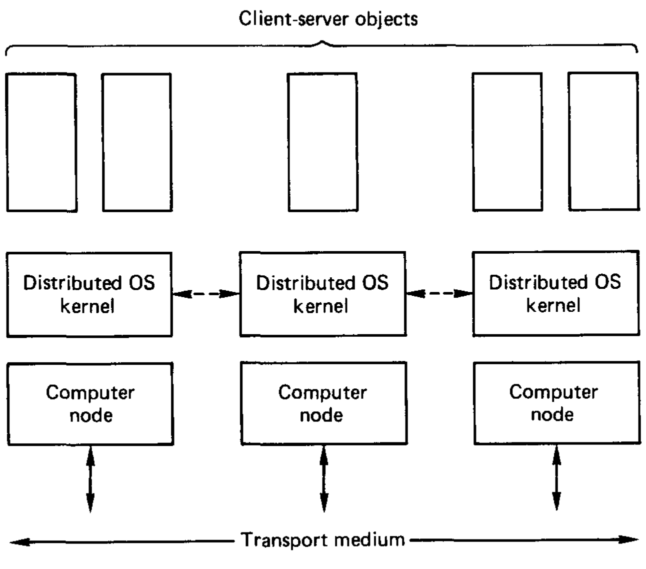

Figure 1. Basic distributed operating system architecture

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 by Hewitt[12]:

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

```
ACTOR ActorTypeName
    Actor variables;

OPERATION OperationName (Parameters)
Local variables;
DO
    Statements;
END
:
:
OPERATION OperationName (Parameters)
Local variables;
DO
    Statements;
END

END ActorTypeName
```

We denote several instantiations A, B, C 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.OperationName (Parameters);
```

Message delay is assumed to be undefined but finite. Multicast messages can be sent to all actors of the same type by issuing:

```
ActorTypeName.OperationName (Parameters);
```

Each actor can temporarily disable the acceptance of messages addressed to a specific operation. We do not assume the existence of communication mechanisms which imply waiting for an unspecified time period as, for example, a receive statement. This message-driven mode allows the scheduling of actor operations as atomic actions: an activated operation always runs to completion before another activation of an operation within the same actor may occur.

INTERACTION PATTERNS BETWEEN CLIENTS AND SERVERS

In this section we study client-server interaction patterns typically observed at the interface between applications and services provided by operating systems. A *service* is considered to be a set of functions which manage a certain type of resources on behalf of client requests. The application domain usually requires that each service has some preestablished quality attributes which guarantee a certain degree of efficiency, responsiveness, availability, etc.

In its simplest form, each service in a system is represented by exactly one single server. Classical, non-distributed operating systems provide the conceptual view of a single-server-per-service system at the call-interface. A call-interface in a *distributed* single-server-per-service system usually determines the specific server which handles the request and sends an appropriate request message to it:

```
CALL_INTERFACE
    Determine server;
    Build request message;
    Send request message to server;
    Wait for results;
    Return results to application;
END
```

Call-interfaces in distributed systems of this structure are commonly organized as stubs which map a procedural view of services to the message view. However, the one-to-one correspondence between services and servers is neither realistic nor desirable in a distributed environment. Requirements for incremental growth of the service performance as well as reliability considerations result in multi-server-per-service structures. Typical examples of multi-server structures found in existing systems are:

1 *A pool of identical servers:* on service request, the selection of an adequate server of the pool can be achieved according to several strategies:

- The client picks an arbitrary server by random choice.
- The client directs its request to a pool manager acting as a service mediator or trader which manages the state of all servers and forwards the request to the most appropriate server.
- The client broadcasts its request to all other clients to coordinate it with potentially competing requests from other clients. Access to a specific server is granted according to a preestablished agreement or election protocol.
- The client broadcasts its request to all servers of the pool. The servers start an internal cooperation protocol to determine the server which should handle the request.

2 *Replicated servers:* a standard way of achieving fault-tolerance in distributed systems is by server replication. Again, a wide spectrum of cooperation protocols exists to enforce consistency and service resiliency in the presence of faults.

3 *Cooperating servers:* a typical example is the processing of a request by a pipeline of servers.

Furthermore, combinations and variants of the abovementioned schemes are possible. It should be noted, however, that because of typical delays in information propagation and acquisition in distributed systems the selection of an adequate server is a non-trivial problem[13]. In general, multi-server services require complex cooperation schemes between the participating clients and servers. Simple stubs are insufficient to handle the abovementioned service types. The discussion shows that a well defined structure representing services in distributed systems is needed.

SERVICE MODEL

In the sequel we assume that from a conceptual point of view every service in a distributed system is represented by a *service manager*, as depicted in Figure 2. The service manager coordinates the servers which contribute to the service and it hides details of the server structure to clients and other servers. Clients always access the service through the service manager. They are unaware of the underlying server structure. *Vice versa*, a server offering its service to the service manager is unaware of specific clients and of other servers contributing to the service. The symmetric *server 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 increased performance, 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

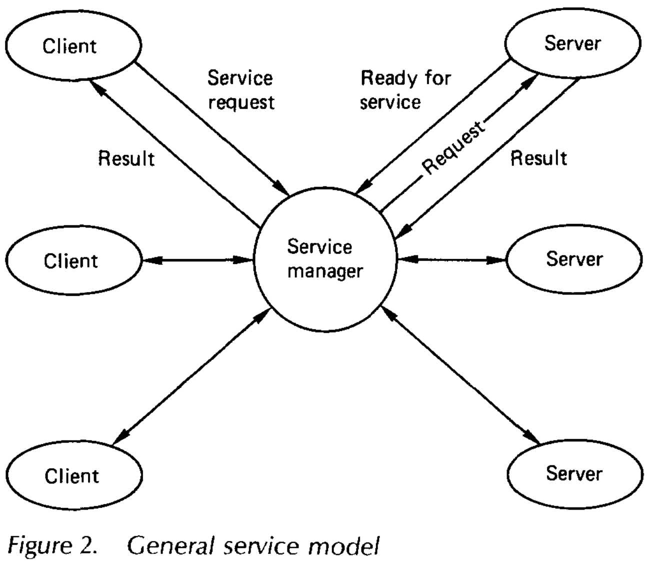

Figure 2. General service model

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

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

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

   OPERATION ReadyForService (Server__ID)
   DO
      IF request pending THEN Server[Server__ID].
                              Request( . . . );
                              ELSE Ready[Server__ID] := true;
      END
   END

   OPERATION 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) has to provide an operation 'Result' for the subsequent delivery of the results by the service manager. Each server offers as many 'Request' operations as there are different functions defined for this type of service. The general structure of a server is:

```
   ACTOR ServerType
      State information;
   :
   OPERATION Request ( . . . )
   DO
      Process the request;
      ServiceManager.Result ( . . .);
      IF ready for servicing further requests
         THEN
            ServiceManager.ReadyForService (own__ID);
      END
   END
   :

   END ServerType
```

It should be clear from the preceeding discussion that the idea of a service manager as depicted in Figure 2 is only a conceptual view. Reliability and performance considerations demand distributed realizations.

ARCHITECTURE FOR IMPLEMENTATION OF DISTRIBUTED SERVICES

In a truly distributed system one strives to avoid centralized control structures to retain the potential benefits of distribution. The logical structure of a service organization as shown in Figure 2 can be mapped onto 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*.

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

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

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 other service agents to achieve agreement on the servers to become involved in the service.
4 The service agent forwards the request directly to the selected servers.

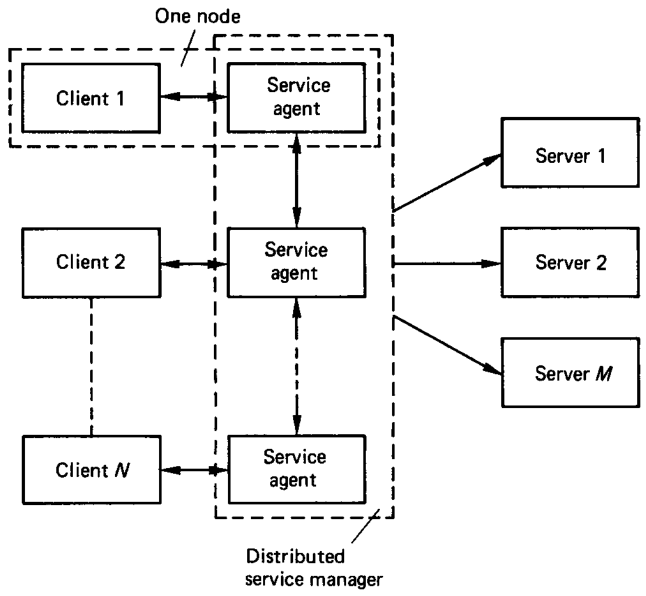

Figure 3. Client-oriented organization of a distributed service

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 mutual exclusion as proposed originally by Lamport[14] is an example of 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 of keeping the local views on the semaphore states consistent. This approach can be employed to synchronize the access to servers if a distributed semaphore is associated with each server.

Alternatively, one can distribute the functionality of the service manager of Figure 2 to a set of service agents associated in one-to-one correspondence to servers. The resulting structure is depicted in Figure 4.

The general *server-oriented* distributed service organization scheme may be described as:

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 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 (or, in simple cases, the server itself) finally returns the result to the initiating client.

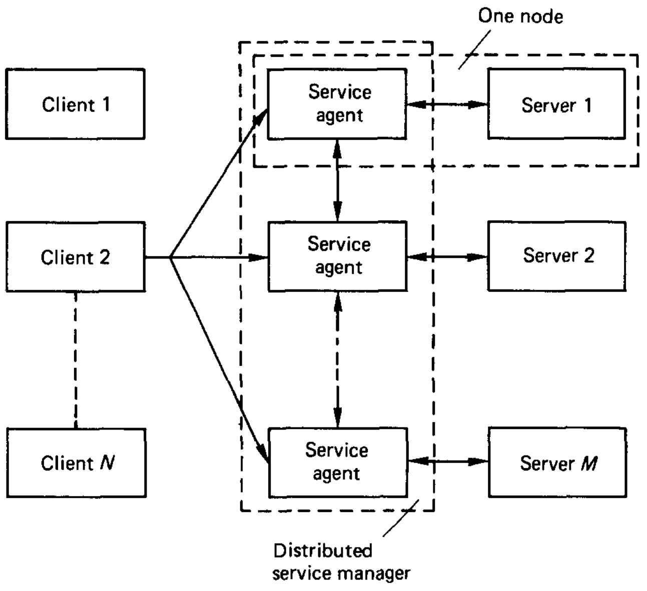

Figure 4. Server-oriented organization of a distributed service

An example of this server-oriented distributed service organization is the management of a pool of identical resources as, for example, printers: a client broadcasts its request to the set of service agents each responsible for managing a particular print server. (Alternatively, the client might address only a single service agent.) 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.

Besides the two extreme organizations for distributed services sketched above there might be good reasons for a *hybrid approach* which combines the advantages of client-oriented and server-oriented organizations for distributed services. Following this idea, the service manager of Figure 2 is distributed into a set of *service agents for clients* (SAC) and a set of *service agents for servers* (SAS). This approach is illustrated in Figure 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 aproach (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;
:
OPERATION ServiceRequest (Parameters)
DO
```

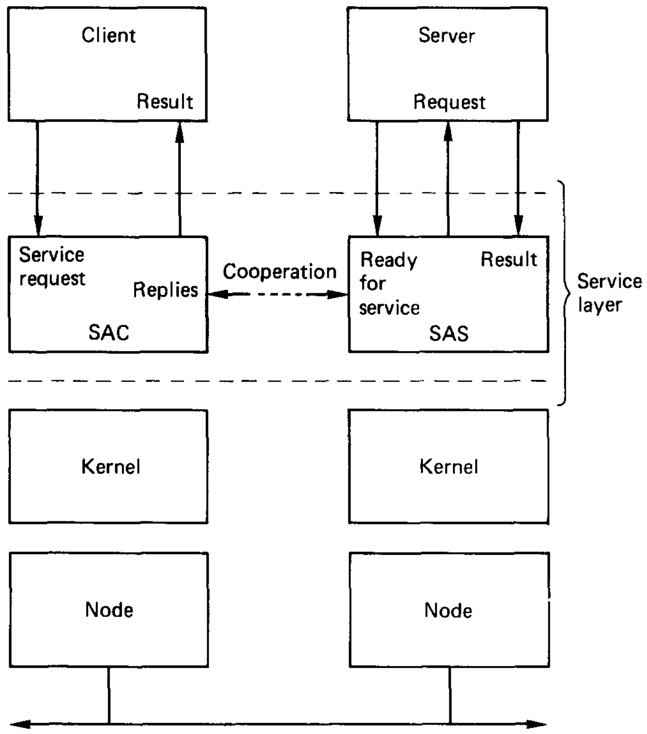

Figure 5. Distributed system architecture with a service layer

```
        Negotiation with other service agents;
        Distribution of the service request to a subset of other
        service agents (SASs);
    END
    :
    OPERATION Replies (result)
    DO
        collect the results;
        client.Result (result); /*return result to client*/
    END
    :
    END SAC
```

```
    ACTOR SAS /*service agent for servers*/
        State information;
        Ready:bool;
    :
    OPERATION ServiceRequest (Parameters):
    DO
        IF not Ready THEN queue the request;
           ELSE
              Server.Request (. . . );
              Ready := false;
        END
    END
    :
    OPERATION ReadyForService
    /*called from associated server*/
    DO
        IF request pending THEN Server.Request ( ... );
                            ELSE Ready := true;
        END
    END
    :
    OPERATION Result ( . . . )
    DO
        /*Send result to responsible SAC*/
        SAC[ . . . ].Replies ( . . . );
    END
    :
    END SAS
```

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 as introduced in the previous section; to a large degree, its structure is transparent to the clients and servers. So as not to obscure the main idea, the examples are somewhat simplistic.

Example 1: 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. For this purpose we introduce the operation 'NewLoad' in the service agent called by the corresponding server. (For efficiency reasons, the current load index might also be piggybacked on other messages sent by the server to its service agent.) 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 or multicast mechanisms using an echo algorithm covering all service agents[15]. For ease of presentation a *virtual ring* going through all service agents is assumed in the example below:

```
    ACTOR SAS_CompServer
        my_ID:. . . /*identification of the current actor*/
        NextSAS: SAS_CompServer;
        CurrentLoad: float;

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

    OPERATION AcceptRequest /*from SAC*/
    DO
        Store caller_ID;
        Disable further AcceptRequests;
        NextSAS.Forward (CurrentLoad, my_ID, my_ID)
    END

    OPERATION Forward (load, minID, origin)
    DO
        IF origin = my_ID
        THEN
            send minID to SAC[caller_ID];
            Enable AcceptRequests;
        ELSE
            IF Currentload <load
            THEN
                NextSAS.Forward (CurrentLoad, my_ID, origin);
            ELSE
                NextSAS.Forward (load, minId, origin);
            END
        END
    END

    OPERATION ReadyForService . . . END

    OPERATION Result . . . 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 only yields an approximation, since the situation might have changed while the inquiry message was circulating. The problem of selecting a server based on outdated information due to delays in information acquisition is discussed by Wolisz[13]. Another drawback of this simple solution is that several clients calling 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. probing only a subset of compute servers, propagating the inquiry message in parallel to the service agents, making a random choice among several appropriate compute 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: 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 Figure 6. Here, the service is organized by two identical copies of the server. It is accessed by three functions: acquire, use and release. To simplify the presentation it is assumed that the following conditions hold:

- only fail-stop failures for servers are considered;
- the probability of having more than one server fail at the same time is negligible;
- the servers are autonomous, i.e. they do not involve other servers to provide their intended services;
- messages do not get lost by the communication medium.

With respect to the general service organizations, as outlined in the previous section, 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 do not show the distributed mutual exclusion scheme for

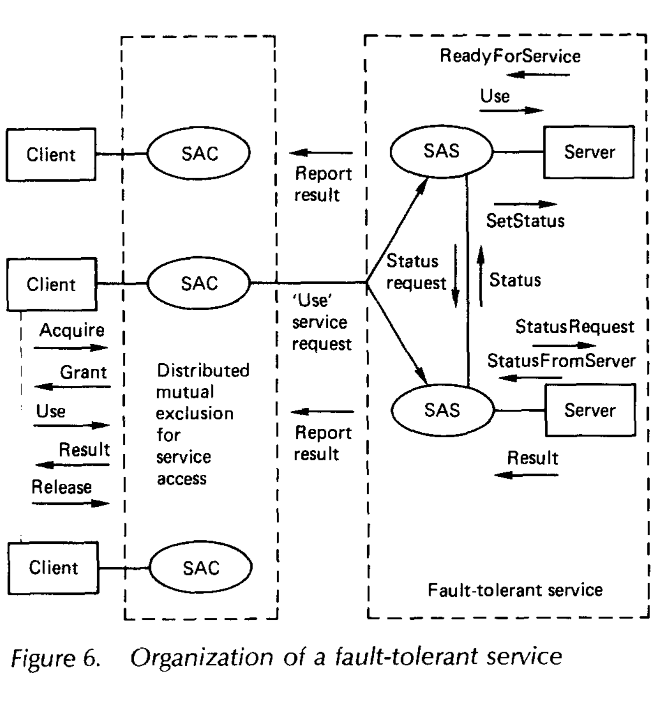

Figure 6. Organization of a fault-tolerant service

'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 simple pattern:

- Each 'use' request is multicast by the responsible SAC to the corresponding SASs.
- 'Use' requests are queued in both SASs.
- Each SAC 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 which issued 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 crashed and later restarted server: it must be assured that the first request message accepted for processing by the restarted server is consistent with its internal state resumed after restart. 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 then those contained in the queue received. Such request messages are kept in the local queue, all other messages in the local queue are removed. (The SAS records the highest sequence number per client which has been processed. Messages which are in the local queue with a higher sequence number 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. It is assumed that when recovering from a crash a server eventually calls the operation ReadyForService of its corresponding SAS after setting RestartPhase to true:

```
ACTOR SAC

OPERATION Acquire . . . END

OPERATION Release . . . END

OPERATION Use (Parameters) /*from client*/
DO
    seq_no := seq_no + 1;
    /*Multicast the request to both SASs*/
    SAS.ServiceRequest (Parameters, seq_no);
END
```

```
OPERATION Report (result) /*from SAS*/
DO
    IF first reply THEN client.Result (result);
      /*return result to client*/
    END
END

END SAC
```

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

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

OPERATION Result (...) /*from server*/
DO
    Report result to responsible SAC;
END

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

OPERATION StatusFromServer (S)
DO
    RemoteSAS.Status (S, LocalQueue)
END

OPERATION Status (S, RemoteQueue) /*from remote SAS*/
DO
    FOR each M in LocalQueue
    DO
      IF seq_no of M < = highest seq_no of
      corresponding client in RemoteQueue
      THEN
        remove M from LocalQueue;
      END
    END
    server.SetStatus (S);
    LocalQueue := LocalQueue + RemoteQueue;
    RestartPhase := false;
    IF #LocalQueue > 0
      THEN
        server.use (...)
        ServerReady := false;
      ELSE
        ServerReady := true;
    END
END
OPERATION ServiceRequest (Parameters, seq_no)
/*from SAC*/
DO
    IF RestartPhase or not ServerReady
      THEN
        buffer request message with
          seq_no and client ID in LocalQueue;
      ELSE
        server.use (...);
        ServerReady := false;
    END
END

END SAS
```

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 server 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 extent can the cooperation schemes be made generic (e.g. regarding actual system configurations)?
- How scalable are the involved algorithms, and how well do they perform in a dynamic environment with a varying number of servers and clients?
- What are useful and required communication and synchronization concepts (e.g. atomic actions, RPC, interrupts)?
- How much knowledge do clients and servers need to have about each other to do their work?
- How much information do service agents need from their corresponding clients and servers in order to perform their task?
- Can server transparency be realized in an efficient way as an autonomous layer on top of an operating system kernel, or should it be realized mostly by library functions called from servers and clients?

Possible answers to these questions have a strong impact on the degree of transparency that can be achieved. These and related problems concerning the structured organization of service layers will be addressed in the future. We hope that eventually it will be possible to integrate the mechanisms and paradigms into a distributed system construction toolkit such as the ISIS system[16].

REFERENCES

1 **Enslow, P** 'What is a distributed processing system?' *IEEE Computer*, Vol 11 No 1 (1978) pp 13-21

2 **Tanenbaum, A S and Renesse, R** 'Distributed operating systems' *ACM Comput. Surv.* Vol 17 (1985) pp 419-470

3 **Cheriton, D R** 'The V distributed system' *Commun. ACM*, Vol 13 No 3 (1988) pp 314-333

4 **Accetta, M, Baron, R, Bolosky, W, Golub, D, Rashid, R, Tevanian, A and Young, M** 'Mach: A new kernel foundation for UNIX development' *Summer USENIX Conf.*, (1986) pp 93-112

5 **Dasgupta, P, LeBlanc Jr, R J and Appelbe, W F** 'The Clouds distributed operating system' *Proc. 8th ICDCS*, (1988)

6 **Mullender, S J and Tanenbaum, A S** 'The design of a capability-based distributed operating system' *Comput. J.*, Vol 29 No 4 (1986) pp 289-300

7 **Ousterhout, J K, Cherenson, A R, Douglis, F, Nelson, M N and Welch, B B** 'The Sprite network operating system', *Computer*, Vol 21 No 2 (1988) pp 23-36

8 **Rozier, M, Abrossimov, V, Armand, F, Boule, I, Gien, M, Guillemont, M, Herrmann, F, Kaiser, C, Langlois, S, Leonard, P and Neuhauser, W** *CHORUS Distributed Operating Systems*, Technical Report CS/TR-88-7.6, Chorus Systems, France (1988)

9 **Schröder, W** 'PEACE: A distributed operating system for an MIMD message passing architecture' *PROC. 3rd Int. Conf. on Supercomputing*, Boston, MA (1988) pp 302-312

10 **Birell, A D and Nelson, B J** 'Implementing remote procedure calls' *ACM Trans. Comput. Syst.*, Vol 2 (1984) pp 39-59

11 **Shatz, S M** 'Communication mechanisms for programming distributed systems' *IEEE Computer*, Vol 17 No 6 (1984) pp 21-29

12 **Hewitt, C** 'Viewing control structures as patterns of passing messages' *Artif. Intell.*, Vol 8 (1977) pp 323-364

13 **Wolisz, A** 'Service provider selection in an open services environment' *Proc. 2nd IEEE Worshop on Future Trends of Distributed Comput. Syst.*, (1990) pp 229-235

14 **Lamport, L** 'Time, clocks, and the ordering of events in a distributed system' *Commun. ACM*, Vol 21 No 7 (1978) pp 558-564

15 **Chang, E J H** 'Echo algorithms - Depth parallel operations on general graphs' *IEEE Trans. Softw. Eng.*, Vol SE-8 No 4 (1982) pp 391-401

16 **Birman, K P, Joseph, T A and Schmuck, F** *ISIS - A Distributed Programming Environment*, Technical Report 87-849, Cornell University, New York (1987)