Proc. Natl. Sci. Counc. ROC(A)
Vol. 19, No. 4, 1995. pp. 309-320

The Design and Implementation of a Distributed Semaphore Facility: DISEM

CHAIO-CHUNG WU, HSIOU-MIAN LIEN, AND SHYAN-MING YUAN

Department of Computer and Information Science
National Chiao Tung University
Hsinchu, Taiwan, R.O.C.

(Received October 21, 1993; Accepted November 24, 1994)

ABSTRACT

Most operating systems provide a semaphore mechanism to users because the semaphore is an excellent, classic mechanism for synchronizing concurrent processes. This paper describes the design and implementation of a distributed semaphore facility called DISEM (DIStributed SEMaphore). DISEM supports a semaphore mechanism in a distributed workstations environment, and it is implemented at the application level for workstations running a version or a derivative of the UNIX operating system which supports both BSD sockets and System V IPCs. The use of the DISEM system is made entirely transparent. It enables use of many of these library functions to be as simple across machine boundaries as within a single machine. In addition, it also provides fault tolerant service in case a workstation crashes. In order to distribute the server load and to enhance the reliability of the system, we distribute the shared data onto several servers. In other words, the distributed semaphores in the system are not managed by a central server, but rather partitioned and served by several servers. The preliminary performance results indicate that the effectiveness of our DISEM system is acceptable. Generally speaking, DISEM is a useful and portable facility for supporting a distributed semaphore in a local area network of workstations.

Key Words: distributed system. interprocess communication, synchronization, distributed semaphore, fault tolerance

I. Introduction

Distributed systems have become more and more attractive in recent years because of the drop in hardware costs and advances in computer network technologies. A distributed application usually contains several concurrently executing processes which may be executed in different hosts. Usually, there is direct or indirect interaction between these processes. To ensure correct cooperation among these concurrent processes, a distributed system must provide mechanisms for process synchronization and communication. There are many problems related to synchronization in a distributed system, for example, serialization of concurrent access to shared objects by multiple processes, synchronization between the sender and receiver of a message, and specification and control of the joint activity of cooperating processes, and so on. Therefore, process synchronization is crucial for the design of a distributed system.

Conventionally, semaphore is a very powerful and classic mechanism for handling process synchronization and mutual exclusion control. Most modern operating systems provide such a mechanism, for example UNIX System V IPCs. However, there are only few commercially available semaphore facilities for distributed systems (Fleisch, 1986). Hence, we design and implement a distributed semaphore facility to support process synchronization and mutual exclusion in a distributed environment.

Distributed systems have several advantages over centralized ones. The major advantages include efficient utilization of resources, the ability to enhance the system gradually, and computation speedup. In addition, an important and desirable property of a distributed system is its ability to tolerate failures (Schneider, 1986). This stems from the fact that most failures in such systems are only partial, and that there may be enough resources left to allow operation to proceed in spite of failures. Therefore, we have introduced a new architecture which best exploits this capability into our *Dis*tributed *Sem*aphore facility *DISEM*. The main advantages claimed for the new architecture are simple strategies and low network traffic overhead for fault tolerance.

The remainder of the paper is organized as follows. In the next section, we will give the definitions of the distributed semaphore that will be used in the remainder of the paper and review other related works. Section III states the overall strategies of our design. In Section IV the detailed implementation of DISEM is described; moreover, the user interfaces and the performance evaluation of our DISEM are presented. Finally, we conclude the paper by emphasizing the important achievements of the project and our plans for future work in the last section.

II. Definitions and Related Works

This section gives some basic definitions for the semaphore and considers some examples to illustrate its usages. Some corresponding works on semaphores are also reviewed in this section.

1. Definitions and Notations

Semaphores are very powerful synchronization primitives. They were first put forward by the Dutch theoretician Dijkstra (1965, 1968) as an alternative means to reduce the complexity involved in achieving mutual exclusion of n processes using busy waiting. Semaphores can be used to prevent two or more processes from accessing a shared resource simultaneously; our distributed semaphore is also used in the same way. The processes can synchronize their operations on the different hosts by using the distributed semaphores.

A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard *atomic* operations: P and V. P and V are also sometimes called "*wait*" and "*signal*", respectively. The classical definitions of P and V are

```
P(S): while S≤0 do skip (nothing)
        S:=S−1;
V(S):S:=S+1;
```

Modifications to the integer value of the semaphore in the P and V operations are executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify the same semaphore value. In addition, in the case of $P(S)$, the testing of the integer value of $S(S \leq 0)$ and its possible modification ($S := S-1$) must also be executed without interruption. The semaphore system calls in the UNIX System V are a generalization of Dijkstra's P and V operations, in that several operations can be done simultaneously, and increment and decrement operations can be done using values greater than 1.

Semaphores can be divided into two classes: *binary semaphores* and *general semaphores.* A semaphore is called a *binary semaphore* if it assumes only binary values 0 or 1. A semaphore is called a *general* or *counting semaphore* if it assumes arbitrary non-negative integer values. Our distributed semaphores are implemented as general semaphores.

2. Usages of Semaphores

Usually, semaphores are used to deal with the n-process critical-section problem. The n processes share a common semaphore, *mutex* (standing for *mut*ual *ex*clusion), initialized to 1. Each process P_i is organized as follows.

```
repeat
  P(mutex);
    critical section
  V(mutex);
    remainder section
until false;
```

In addition, semaphores can also be used to solve various synchronization problems. For example, consider two concurrently running processes: P_1 with a statement S_1, and P_2 with a statement S_2. Suppose that we require that S_2 be executed only after S_1 has been completed. We can implement this scheme readily by letting P_1 and P_2 share a common semaphore *synch*, initialized to 0, and by inserting the statements

```
S1;
V(synch);
```

in process P_1, and the statements

```
P(synch);
S2;
```

in process P_2. Since *synch* is initialized to 0, P_2 will execute S_2 only after P_1 has invoked $V(synch)$, which is after S_1. In addition, to control resource allocation, general semaphores are useful. The initial value of a semaphore S can be set to be the number of units of resource available; $P(S)$ or $V(S)$ is, respectively, executed when a unit of resource is allocated or is released.

Furthermore, semaphores can be used to solve some classical synchronization problems, such as the bounded-buffer (producer-consumer) problem, the readers-writers problem, and the dining-philosophers problem. These problems are important because they can be modeled for a large class of concurrent control problems. An example of using a distributed semaphore to resolve the dining- philosophers problem will be given in Appendix.

3. Related Works

In distributed systems, the implementation of mechanisms for communication and synchronization has traditionally taken one of two approaches: the application-level based approach and OS (Operating System) kernel-level based approach. The distributed System V IPCs in Locus (Fleisch, 1986) is one of the early attempts in realizing a distributed semaphore mechanism. It is an OS kernel-level implementation and concerns itself with augmenting the Locus distributed operating system.

An important issue in any distributed shared memory system is synchronized access to shared data. Therefore, there are many distributed shared memory systems which use distributed semaphores to synchronize access to shared data. The implementation of distributed shared memory on a hypercube as explained by Li and Schaefer (1989) guarantees coherence of shared memory using message passing to provide distributed semaphores for synchronization. Their algorithms allow ownership of semaphores to change dynamically; that is, semaphores can migrate to the sites where they are accessed. However, this design decision is based on low latency of message transmission on Intel iPSC/2 hypercube multicomputers. Clouds provides the distributed semaphore mechanism to support the abstraction of a distributed shared memory, as described by Khalidi *et al.* (1989). Unlike Li's scheme, which uses the dynamic approach, Clouds's semaphore operations are performed statically at the owner site.

However, the usefulness of the systems described above is limited because they are tailor-made for specific environments and require modification of the underlying kernels. Currently, most of the popular engineering workstations are being marketed with one or both of the predominant UNIX systems (BSD and System V). For the foreseeable future, we think UNIX will continue to be the operating system preferred by most users. Since new facilities for distributed and parallel processing are also being developed, to require changes to the kernel has become a costly and self defeating. An alternative is to develop these facilities entirely at the application level.

Consequently, the DISEM design requires no changes to the UNIX operating system kernel. DISEM is implemented entirely at the application level, so it has the advantage of portability for all workstations running a version or a derivative of the UNIX operating system which supports both BSD sockets and System V IPCs.

III. Design Issues

In this section, we describe the strategies used to implement our distributed semaphore facility. In our strategies, a high level of fault tolerance is achieved by redundantly storing data in two independent sites. The main advantages claimed for our strategies are the relative simplicity of the strategies and reduction of overhead due to fault tolerance.

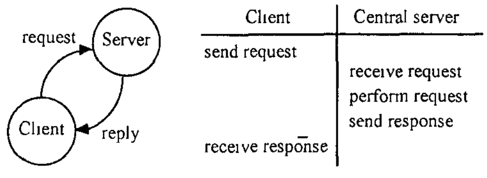
1. Descriptions of Our Approach

A semaphore can be thought of as a shared variable. However, for (loosely coupled) distributed systems, no physically shared memory is available to support such a mechanism. Therefore, we must provide an abstract shared memory in the distributed system to simulate a shared physical memory for our distributed semaphore. That is, some location in the network implements the physical memory, and all processes access the memory through the network operations of the underlying communication system.

As stated above, a distributed semaphore can be treated as a special type of distributed shared memory. The simplest strategy for implementing distributed shared memory uses a single central server which is responsible for serving all accesses to shared data. The central server, or data server called later, maintains the one and only copy of the shared data. Our distributed semaphore algorithm follows this approach. The semaphore operations (*P* and *V*) involve the sending of a request message to the data server by the process executing of the operation, as depicted in Fig. 1(a). The data server accomplishes the request and responds with a reply to the client. In our model, each site runs the same and only copy of DISEM. Meanwhile, only one site of the DISEM is the central server while the others are clients.

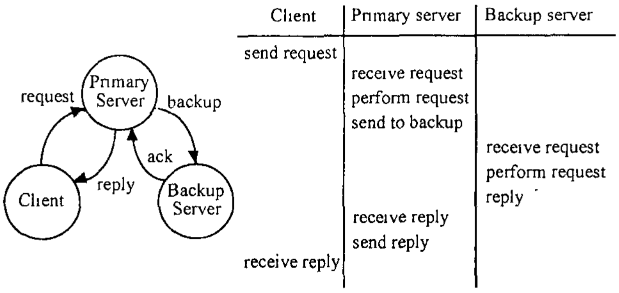
A simple request-response protocol can be used for communication in implementation of this algorithm. For communications reliability, a request is retransmitted after each time-out period when there is no response from the server. For each of semaphore operations, the server must keep a sequence number for each client so that it can detect duplicate requests (due to retransmissions) and acknowledge them appropriately. In our model, a failure condition is detected after several time-out periods with no response.

One potential problem with the central server is that it may become a bottleneck since it has to serve requests from all the clients. In order to distribute the



Client	Central server
send request	
	receive request
	perform request
	send response
receive response	

(a) The base algorithm



Client	Primary server	Backup server
send request		
	receive request	
	perform request	
	send to backup	
		receive request
		perform request
		reply
	receive reply	
	send reply	
receive reply		

(b) The fault tolerant algorithm

Fig. 1. The central server algorithm.

server load and to enhance the reliability of the system, we distribute the shared data onto several servers. In other words, all distributed semaphores in the system are not managed by a central server but are partitioned and served by several servers. We will explain in detail how the server's load is distributed in the next subsection.

2. Fault Tolerant Features

In order to make the distributed semaphore mechanism more reliable, some degree of fault tolerance is a mandatory requirement of such distributed facilities. In a computing environment consisting of a local area network of workstations, a workstation may incur errors due to hardware or software failure. Software failure is caused by incorrect programming or even ill-devised algorithms. Errors of this kind are very unpredictable and hard to deal with as they contradict the assumption that all the execution results will preserve the consistency of the system. In general, applications use software design such as N-version programming (Avizienis, 1985) and the recovery block approach (Randell, 1975) to handle software failure. Therefore, we do not handle such errors. On the other hand, hardware failure such as site crashes are more predictable than software failure. Thus our fault tolerant approach is aimed primarily at hardware failure.

As mentioned above, we just tolerate hardware failure in our model. Instead of trying to survive all types of hardware failure, we attempt to survive only the most common types (site crashes) and attempt to do so at a reasonable cost. Since we observe that site crashes in a distributed system are usually independent of each other and are relatively infrequent (say, at most several a day), we have designed our fault-tolerant algorithm such that it will tolerate only single site crashes. As soon as a site crash is detected, the algorithm enters a *recovery phase*, after which they are again capable of tolerating another single site crash. 1-resilient algorithms appear desirable because the cost of providing this degree of resiliency is relatively modest (compared to more resilient versions of the algorithm) and because the time it takes to recover from a failure can be made short, so that the probability of a second fault occurring during the recovery phase is small. The cost of *t*-resilient algorithms with $t>1$ is substantially higher and appears to be acceptable for only the most critical applications. Therefore, we only consider the case of a single site crash (detected through timeouts and taking on the order of a few seconds in an Ethernet-based environment) and assume that sites act in a fail-stop way (Schlichting and Schneider, 1983); that is, a failed site completely halts in response to a failure in such a way that processes on other sites can detect that a failure has occurred.

Replicating the data on different nodes is the key to data fault tolerance. According to Yap *et al.* (1988), the distributed schemes (in contrast to non-distributed ones (Bartlett, 1981)) that use replication to support fault tolerance can be divided into two categories: the primary-standby approach and the modular redundancy approach. The primary-standby approach (Walker *et al.*, 1983; Birman *et al.*, 1984; Borg *et al.*, 1984) selects one copy from the backups and designates it as the primary one whereas the others are standbys. Then all subsequent requests are sent to the primary copy only. The standby copies are not responsible for this service, and they only synchronize with the primary copy periodically. In the case of a failure, one of the standby copies will be selected as the new primary one, and service goes on from the last synchronized point. Such a scheme does not waste resources on duplicated copies although recovery from a crash is not instantaneous. A variation is primary-secondary copies (Popek *et al.*, 1981), in which the modification requests are propagated to the secondary copies, but consistency between secondary copies is not guaranteed at any time, and enquiries are not limited to the primary copy.

The modular redundancy approach (Banino *et al.*, 1985), also called unanimous update (Bernstein and Goodman, 1984), makes no distinction between the primary copy and the standby ones. Requests are sent to all the backups simultaneously, and service is performed by all the copies. Thus, as long as there exists at least one correct copy, fault tolerance is provided. In contrast to the primary-standby approach, service continues instantaneously after a fault occurs, but it is costly to maintain synchronization between the duplicated items, especially when there are many of them.

Basically, our fault tolerance model, which we call the "host-standby" strategy, is a form of the modular redundancy approach. Each distributed semaphore has its own hot-standby replica maintained by the backup server at the other site, as depicted in Fig. 1(b). Other methods of achieving fault tolerance were considered to be too costly in terms of run-time delays. Our goal has been to achieve as much simplicity as possible. Elegant but complex solutions are generally costly in terms of delay. In the following, we will show how the model works.

Under normal circumstances when failures are absent, a service request is sent to the primary server of the distributed semaphore. Upon receiving a request, the primary server of the distributed semaphore propagates the request to the backup server of the distributed semaphore. While the primary server waits for acknowledgement from its backup server after it has sent out the request, it also handles the request. Upon receiving the ACK from its backup server, the primary server sends the result to the requesting client if the request has been completely processed. From the viewpoint of client, however, the request is made only once, and only one copy of the result is received by the client. Figure 2 shows message flow for the client's request in normal condition.

We now informally argue the fault tolerance of our model by considering the failure of individual processes at different points in time and by describing the steps necessary to recover from failures. Let us examine our model under the following conditions.

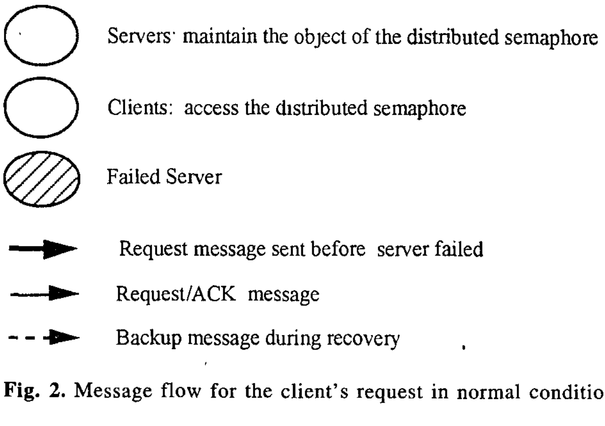

Primary Server
1
4
2
3
Client
Backup Server
Servers: maintain the object of the distributed semaphore
Clients: access the distributed semaphore
Failed Server
Request message sent before server failed
Request/ACK message
Backup message during recovery

Fig. 2. Message flow for the client's request in normal condition.

First, if a client site fails, then either the data server (primary/backup) received the client's most recent request, in which case the system behaves as if the client failed immediately after the request had been executed, or the data server (primary/backup) did not receive the most recent request, in which case the system behaves as if the client failed without having executed this request. We usually do not handle such failures, because they will not affect the normal operations of the entire system. That is, the failed site will not be the server (primary/backup) of any distributed semaphore in the system. However, there is an exception if the failed client obtains some distributed semaphores (i.e. *P* operations) and did not release them (i.e. *V* operations) so that other active clients cannot obtain these distributed semaphores. The primary server will then detect that the client site has failed by repeatedly receiving the same request from other active clients, and then the system will perform recovery for this failed client site.

The second case is when failure occurs in the primary server. If the primary server site fails, then the backup server becomes the new primary server, and it creates and starts a new backup server from hosts which are using the distributed semaphore and passes a copy of the shared data to it. This kind of failure can occur at two times: before or after the primary server has propagated the request. In the event that the failure occurs in the primary server before the request is propagated to its backup server, the client will detect that the primary server has failed by repeatedly timing out when it tries to send the same request. As shown in Fig. 3, after recovery actions are completed, the client will send the same request to the new primary server. Then everything will continue just as if no error has occurred except that the client will resend the request, and the result will be sent from the new primary server to the client.

Now, let us consider the case of failure occurring in the primary server after the request is propagated to its backup server, as shown in Fig. 4. If the primary server fails after the request has propagated to its backup server, in the same manner as stated above, the client will detect that the primary server has failed by repeatedly timing out when it tries to send the same request. After recovery actions are completed, the client will send the same request to the new primary server (backup server). The new primary server will receive the same request twice. Since each request carries a sequence number, the new primary server will process the request only once. And if the new primary

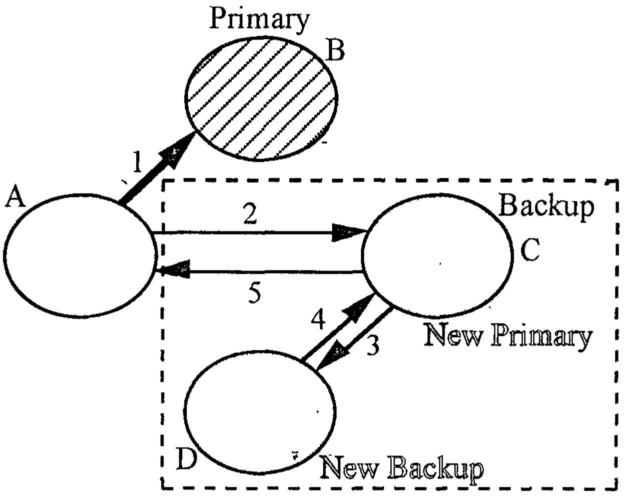
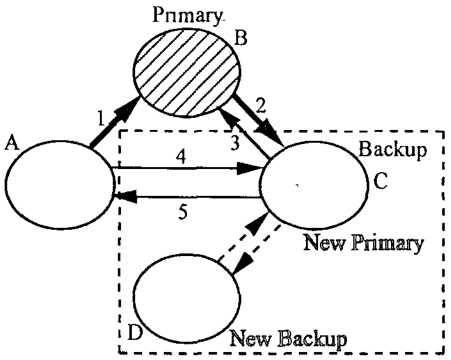
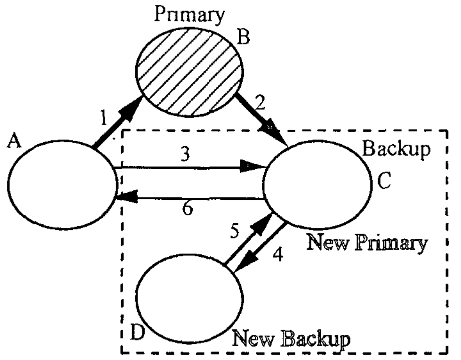


Fig. 3. Message flow for the client's request when the primary server fails before it does not send this client's request to its backup server.



(a) backup server has done the client's request before recovery



(b) backup server does not do the client's request before recovery

Fig. 4. Message flow for the client's request when the primary server fails after it has sent this client's request to its backup server.

server has processed this request before recovery, the result will be sent directly from the new primary server to the client; otherwise the new primary server will have to propagate this request to its new backup server (as in the normal case) and return the result of this request to the client when it receives the acknowledgement from its new backup server. The fact that a new site now serves as the primary data server causes a minor addressing problem: clients will address their next request to the failed primary server and, therefore, will timeout. Our solution to this problem is that as soon as a new primary server is created, it broadcasts its address to all the clients in the system.

In case the backup server site fails, the primary server will detect this (when it receives the same request repeatedly from the client but finds this request has been done by itself), start up a new backup server from hosts which are using that distributed semaphore and pass a copy of the shared data to it, as shown in Fig. 5. If the backup server fails while serving a request, then after recovery the new backup data copy will reflect that change. This is because the new backup data copy is a hot-replica of its primary server's data space that has been changed. After recovery actions are completed, the client will send the same request to the primary server. Then everything will continue just as in under normal conditions except that the client will resend the request.

There are some problems involved in our model. First, how do we select a data server as the primary server of a distributed semaphore in the system? We choose the simplest and most effective way: let the host which first uses the distributed semaphore be the primary server of that distributed semaphore in the system. Secondly, there is the similar problem of how we select a data server as the backup server of a distributed semaphore. We let the host which next uses the dis-

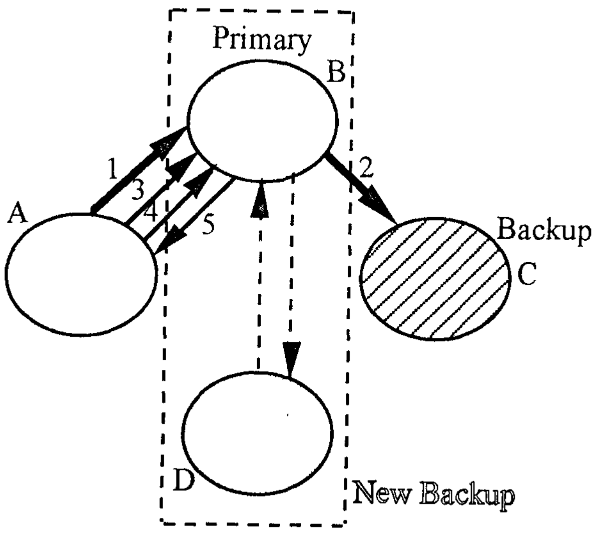


Fig. 5. Message flow for the client's request when the backup server fails.

tributed semaphore be the backup server of that distributed semaphore in the system. In this way, performance is optimized because operations on the semaphore are local operations for clients in the primary server.

3. Recovery

Recovery from failure is important for a fault tolerant system. In contrast to fault detection and assessment, which are passive because they are not intended to change the system (Anderson and Lee, 1988), recovery does produce changes in the system in order to accomplish the restoration; therefore it is active. Our recovery procedure has at least two responsibilities. One is to reverse the effect of every semaphore operation which the processes have carried out at the failed site if necessary. The other is to make a new standby replica for the failed semaphores so as to ensure that a hot-standby replica is alway available at the other site in the system.

Dangerous situations could occur in the event that a process carries out a semaphore operation, presumably thereby locking some resource, and then exits without resetting the semaphore value. Such situations can occur as the result of a site-crash or receipt of a signal that causes sudden termination of a process. In this case, at a later time, other processes would find the semaphore locked even though the process that had locked it no longer existed. To avoid such problems, it is necessary to back out the updates so that changes are invisible. It is worthy of notice that we are *only* concerned about site crashes here. To implement this feature, we maintain a list which is called an *undo_list* for every distributed semaphore in the system. The undo_list structure will be discussed more fully in Section IV.2.

In our description, operations on distributed semaphores can involve up to two different sites. The *using sites* are the client machines that issue the request and receive the result; the *storage sites* are the places where a distributed semaphore is actually stored. The storage sites are the (primary/backup) server machines that serve requests from clients.

When a node fails, all other active nodes have to take part in recovery so that the system can continue to function. Specifically, the nodes that are storage sites for the failed semaphores and storage sites whose semaphore values have been updated by the clients at the failed node take part in recovery. When a node i detects that a node N_k has failed, it announces the event to all other nodes in the system and executes the procedure *perform-recovery*$_i$ for the failed node. The procedure, as shown below, is executed by a node i every time the failure of some node N_k is detected or announced.

```
procedure perform-recovery_i for N_k
begin
    (* PART 1*)
    for each using semaphore S_j of site i do
      AVAILABLE[S_j]:=False;
      resident_number[S_j]:=
      number of storage site for S_j;
    (* PART 2; only for storage sites *)
    for each semaphore object S_l of site i do
      if (N_k is the using site of semaphore S_l)
        undo the operations from N_k;
      endif
      if (N_k is the storage site of semaphore S_l)
        change primary_server(S_l) to site i;
        elect the other using site of semaphore S_l to
        act as backup_server(S_l);
      endif
      broadcast a success message of semaphore S_l;
end
```

The boolean variable AVAILABLE of semaphore S_j is used to record the current status of a distributed semaphore, thereby indicating whether the distributed semaphore S_j has recovered completely or not. The *resident_number* of semaphore S_j is the current number of storage sites exclusive of the failed site for semaphore S_j, and its value is either 1 or 2. This information is used to reset the boolean variable AVAILABLE later. In part 1 of the recovery procedure, the fact that semaphore S_j was unavailable is recorded. When a user requests distributed semaphore S_j whose AVAILABLE is False, the request will be blocked until its AVAILABLE becomes True. In addition, we also compute the value of resident_number for each semaphore S_j.

Part 2 contains actions to be performed by the nodes that are the storage sites of some distributed semaphores. If the failed node N_k is a using site of semaphore S_l, then node i uses the information in the undo structure of semaphore S_l to undo the operations of the failed node N_k. If the failed node N_k is a storage site of semaphore S_l, then node i becomes the primary server of semaphore S_l. And, if there exist other using sites of semaphore S_l in the system, then node i will elect randomly one of these using sites of semaphore S_l to act as the new backup server of semaphore S_l. Finally, for every semaphore S_l, node i will broadcast a success message with the primary/backup server's address to all nodes in the system. On the other hand, after receiving this message on each node, the resident_number of semaphore S_l will be decreased by

1 if this node is a using site of semaphore S_l. Once the resident_number of semaphore S_l becomes zero, the boolean variable AVAILABLE of semaphore S_l is reset to True, and this node unblocks the requests concerning semaphore S_l.

IV. Implementation

We have implemented an initial version of DISEM on a cluster of about ten Sun workstations connected by a 10Mbps Ethernet. The DISEM implementation executes on top of UNIX (as a user process). We made no changes to the operating system kernel. This section outlines the implementation of DISEM.

1. DISEM's Architecture

Each workstation involving distributed semaphores runs the same and only copy of DISEM. A DISEM is constructed by two modules: the semaphore server and the communication server. The semaphore server is responsible for providing the main function of DISEM, for maintaining the objects of distributed semaphores, and for sending and receiving messages from all the clients in its site. All intrasite communications are based on UNIX System V IPC - Message Queue. The communication server is only responsible for receiving all messages from the socket on the LAN, and it then relays these messages to the semaphore server through the message queue.

All inter-site communications are based on the SOCK_DGRAM protocol (Leffler *et al.*, 1983). Figure 6 shows the architecture of DISEM.

2. Semaphore Object Management

In our DISEM, we provide a mechanism which is called a *storage_table*. Each entry in the storage_ table stands for a distributed semaphore's object and is implemented as follows:

```
type semaphore=record
        value: integer;
        L: proceess_list;
        U: undo_list;
end;
```

Each semaphore has an integer value and a process_list. When a process must wait on a semaphore, it is added to the process_list. A *V* operation removes one process from the list of waiting processes and awakens it. One method that we use to add and remove the process from the list, which ensures bounded waiting, is first-in first-out (FIFO; a queue), where the semaphore contains both the head and tail pointer of the queue. This scheduling algorithm is fair as no starvation can occur. In addition, there is also a list of undo structure (undo_list) which is stated in section III. 3. Each entry of this undo_list is an *undo* structure that indicates how to undo the semaphore value if a process has updated the semaphore value and is aborted by site crash. The undo structure consists of three components: the hostname of this process, identifier of this process, and adjustment value for this process. Because semaphore does not survive site crashes, this undo structure can be stored in memory.

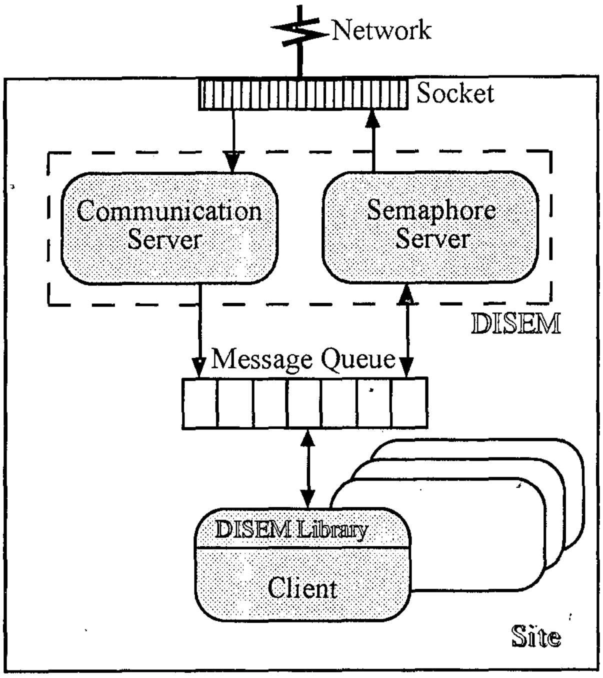


Fig. 6. DISEM's architecture.

Our system allocates undo structures dynamically when a process invokes a new distributed semaphore for the first time. On subsequent semaphore operations (*P* and *V*), we search the process' undo structure for the semaphore with the same hostname and process ID from the semaphore operation: if we find one, we subtract the value of this semaphore operation from the adjustment value. Thus, the undo structure contains a negated summation of all semaphore operations the process had carried out on the semaphore. If no undo structure for the semaphore exists, we create one. The undo structure is removed when the semaphore is closed by the process.

The critical aspect of semaphores is that they are executed atomically. We must guarantee that no two processes can execute *P* and *V* operations on the same semaphore at the same time. This situation is a critical-section problem. However, because our model is the central-server model, synchronization is achieved via mutual exclusion performed by this central server.

3. Name Space Management

A naming service, which binds global and high-level names to objects, is a key component in distributed systems. A global naming service can provide names for objects in the system that can be passed between clients without change in interpretation. In our distributed semaphore, we want to provide a global naming service to meet the above requirement. More importantly, the transparency constraints of the network must be satisfied.

In UNIX System V, IPCs use two types of names: keys and identifiers (Bach, 1986). A key is a 32-bit integer which the user selects and associates with a message queue, a semaphore set, or a shared memory segment. Identifiers operate on objects. Typically when an object is located using a key, an identifier is returned. The name space for our distributed semaphore simulates in the same manner.

In order to implement much of the naming functionality described, we provide a mechanism called *using_table* to locate distributed semaphores whose objects are distributed among some storage sites (servers). Each using site (client) maintains a small using_table that identifies the storage site for a using distributed semaphore. Using_tables are constructed using a simple broadcast protocol and are updated automatically as the configuration of storage sites changes (after recovery). So there is no need for a separate name server in the system and no need to worry about the reliability of the name service.

In our DISEM system, the name *lookup* operation invoked by users (with a create/open call) must return two things from the server: a storage site's address and a token. The storage site's address is used to send requests to the appropriate site, and the token is passed to the site (as part of requests) to identify the object of the distributed semaphore being manipulated. In this case, the token is typically an index into the distributed semaphore table (storage_table) of the storage site; it saves the server of storage site from having to retranslate the name of the distributed semaphore on each request.

Each entry in the using_table corresponds to one of the distributed semaphores being used in the local processes: it contains the name of the distributed semaphore (key), the address of the storage site, and a token. Initially, each using_site starts with an empty using_table. When a user requests a lookup operation on a distributed semaphore, the DISEM of using_site searches the using_table for the name that matches the name of the distributed semaphore. If it finds no matching name in its table, it broadcasts the request to all servers. Each DISEM of storage_site then searches its local storage, and one successful DISEM (if any) responds to the request with the address of the storage site and the token for the distributed semaphore. Then, the DISEM of using_site uses the response to create a new entry in its using_table. Finally, the result of the lookup operation returns the identifier (an index into its using_table) to the user. The identifiers serve the same purpose as the tokens described above: they allow the DISEM of using_site to locate the entry of using_table for a distributed semaphore without having to repeat an expensive name search. If there is no response to the broadcast after a time-out period, then this implies that the distributed semaphore does not exist in the system, and different operations are then carried out depending upon the request. Entries are added to the using_table only when needed: a distributed semaphore that has never been accessed by local processes will not appear in the using_table.

As stated above, a distributed semaphore's identifier is returned as the result of a lookup operation. Application programmers typically do not inspect the identifier. Rather, they present it to the DISEM to access an underlying object. Most function calls we will later describe take an identifier as their first argument. Note however that it is possible that access to a distributed semaphore can be obtained by "guessing" or "misusing" an identifier. Thus, in addition to those mechanisms stated above, we must provide an additional authentication mechanism for access of the identifier to prevent unauthorized access to an identifier from interfering with the normal operation on this semaphore.

This authentication mechanism is called *private_table*. There is a private_table for every user process that involving distributed semaphores. Initially, each user process starts an empty private_table. When a lookup operation returns an identifier, we allocate a new entry of private_table used to store the identifier and reply the index's value of this entry but real identifier to the user. That is to say, the identifier obtained by the user is actually an index to its private_table. In this manner, we can quickly achieve authentication of this distributed semaphore's identifier because if the identifier is valid for the user, we will obtain the real identifier through the index of the user's private_table; otherwise, we will not. Totally, there are three kinds of tables (private_table, using_table, and storage_table) used to access a distributed semaphore, which are depicted in Fig. 7.

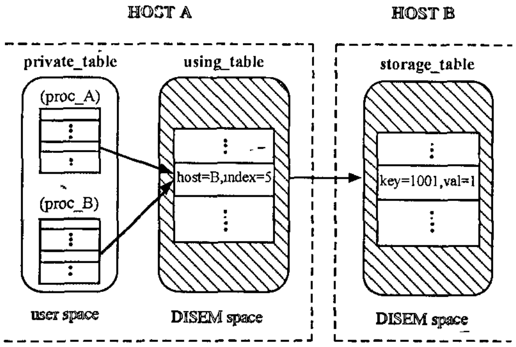


Fig. 7. Data structures for access to distributed semaphore.

4. User Interface

Use of the DISEM system is entirely transparent. DISEM enables users to use much of these library functions as simply across machine boundaries as within a single machine. The system provides access facilities that are invoked in the user's program by calling a set of C functions such as dsem_create(), dsem_open(), dsem_P(), dsem_V() and dsem_close().

A new distributed semaphore with a specified initial value is created by using the dsem_create() function:

int dsem_create (key,val,flag).

dsem_create() returns the distributed semaphore identifier associated with *key*. A semaphore identifier and associated data structure and a distributed semaphore are created for *key* if *key* does not already have a distributed semaphore associated with it. Upon creation, the value of the new distributed semaphore is initialized as *val*. *Flag* is used to indicate whether fault-tolerance is necessary or not for this distributed semaphore.

We have another function, dsem_open(), which should be used instead of dsem_create() if the calling process knows that the semaphore already exist:

int dsem_open (key).

dsem_open() returns the identifier of the existing semaphore associated with *key*. A semaphore identifier is returned for *key* if *key* does already have a distributed semaphore associated with it.

Once a distributed semaphore is created (or opened), operations are performed on the semaphore value using the dsem_P() and dsem_V() functions:

int dsem_P (semid,val,flag)
int dsem_V (semid,val,flag).

dsem_P() is used to perform a general Dijkstra's *P* operation on the semaphore associated with the semaphore identifier specified by *semid*. *val* is the amount of decrement. If the semaphore value is greater than or equal to the absolute value of *val*, the absolute value of *val* is subtracted from the semaphore value. Otherwise, *dsem_P*() will suspend execution of the calling process until the following condition occurs: the semaphore value becomes greater than or equal to the absolute value of *val*. When this occurs, the absolute value of *val* is subtracted from the semaphore value, and the absolute value of *val* is added to the calling process' adjustment value for the specified semaphore.

dsem_V() is used to perform a general Dijkstra's *V* operation on the semaphore associated with the semaphore identifier specified by *semid*. *val* is the amount of increment. The absolute value of *val* is added to the semaphore value, and the absolute value of *val* is subtracted from the calling process' adjustment value for the specified semaphore. *Flag* is used to indicate whether undo is necessary or not for both the dsem_P() and dsem_V() functions.

dsem_close() is used to close the using semaphore associated with the semaphore identifier specified by *semid*:

int dsem_close (semid).

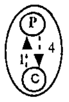
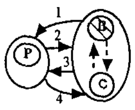
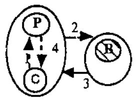
dsem_close() decreases the count of the processes using the semaphore associated with the semaphore identifier specified by *semid* in the system, and if the calling process is the last one, then the function will remove the semaphore from the system and destroy the data structure associated with the semaphore. This function is for a process' to call before it exits, when this is done with the semaphore.

5. Performance Measurements

We measured the performance of the DISEM implementation on UNIX. All measurements were done on SUN 4/40 workstations connected through a 10-megabit per second Ethernet local network. Table 1 presents the average time in milliseconds required for DISEM semaphore operations. The times shown are for the elapsed time (returned by *gettimeofday* system call).

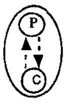
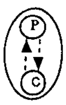
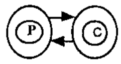
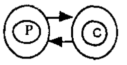
These operations were executed under three different kinds of conditions, which include all the cases of DISEM operation. The first column is the case where the semaphore is local and does not have a backup on the other site. Because all communications are local, it is the most non-expensive one. The second column is the case where the semaphore is remote and has a

Table 1. Measurements of DISEM Operations on UNIX

	 without backup	 with backup	 with backup
dsem_open	3.039 ms	30.529 ms	16.834 ms
desm_P	2.065 ms	29.173 ms	15.637 ms
desm_V	1.842 ms	29.111 ms	15.526 ms

backup on the other site. This case is also a general condition for DISEM operation, and has four steps (request, backup, ack, and reply). It requires four messages on the network and is the most expensive one. The last column is the case where the semaphore is local and has a backup on the other site. Because it requires two messages on the network, its elapsed-time is nearly half as much as that for the second case. The results indicate that the amount of network traffic is a strong influence on the overhead of these operations. Table 2 shows the elapsed times of these primitives implemented by a SUN RPC server which used the UNIX System V semaphores. Because the current SUN RPC server was constructed as an iterative server, the clients had to use busy waiting to avoid causing the server to be blocked. Thus, a successful P operation could actually involve several requests to the RPC server. In this experiment, this re-request effect was ignored. Other shortcomings of RPC implementation are that the centralized server may become a performance bottleneck, and that it suffers from the single point of failure. Although the performance of simple RPC implementation is better than that of DISEM implementation for remote operations, DISEM provides fault-tolerance, nonbusy waiting clients, better performance for local operations, and load balancing. Therefore, the effectiveness of DISEM is clear.

Table 2. Comparisons between DISEM and SUN RPC Server

	 DISEM	 RPC	 DISEM	 RPC
P operation	2.065 ms	3.178 ms	15.482 ms	3.242 ms
V operation	1.842 ms	2.844 ms	15.298 ms	3.560 ms

V. Conclusion

This paper has outlined the objectives and significance of a distributed semaphore facility-DISEM. We have overviewed issues related to DISEM and described DISEM implementation. Currently, the DISEM system has been developed entirely at the application layer, which is highly portable.

The fault tolerant distributed control strategy described in this paper is useful in various situations to realize fault-tolerant global control of resources in distributed computing systems. Fault tolerance is achieved by providing a hot-standby replica at the other site for each distributed semaphore. In order to distribute the server load and to enhance the reliability of the system, we distribute the shared data onto several servers. In other words, all distributed semaphores in the system are not managed by a central server but are partitioned and served by several servers. The user interface to the DISEM system is both simple and clear. In the development of some typical asynchronous concurrent programs, there was no ambiguity or difficulty in using the DISEM system. Our performance results show that the effectiveness of our DISEM system is acceptable.

Future works include development of other distributed interprocess communication facilities such as a distributed message queue and distributed shared memory to form the package of distributed System V IPCs. Also, future work is needed to improve protection of access, management of replicas, and interworkstation communication efficiency.

Acknowledgments

1. The authors would like to thank the referees for their helpful comments. 2. This work ws supported by the National Science Council, under contract number NSC82-0408-E009-053.

Appendix

The following program is the code of *philosopher i* (*i*=0) for the dining philosophers problem.

```
#include "dsem.h"
#define FORK0     (key_t) 1001
#define FORK1     (key_t) 2002
#define UNDO      1
main()
{
  int fork0, fork1;
  fork0=dsem_open(FORK0);
  fork1=dsem_open(FORK1);
  for(;;) {
    sleep(5);          /* Think */
    dsem_P(fork0,1,UNDO);
    dsem_P(fork1,1,UNDO);
    sleep(6);          /* Eat */
    dsem_V(fork0,1,UNDO);
    dsem_V(fork1,1,UNDO);
  }
  dsem_close(fork0);
  dsem_close(fork1);
}
```

Note: We use the dsem_open() function in the program (instead of the dsem_create() function) provided that there exists some *philosopher j* in the system which is responsible for creating the distributed semaphore.

References

Anderson, T. and P. A. Lee (1988) *Fault Tolerance-Principles and Practice*. Prentice-Hall.

Avizienis, A. (1985) The N-version approach to fault tolerant software. *IEEE Trans. on Software Engineering*, **SE-11**(12), 1491-1501.

Bach, M. J. (1986) *The Design of the UNIX Operating System*. Prentice-Hall.

Banino, J. S., J. C. Fabre, M. Guillemont, G. Morisset, and M. Rozier (1985) Some fault-tolerant aspects of the chrous distributed system. IEEE Conference on Distributed Computing Systems, 430-437.

Bartlett, J. (1981) A NonStop kernel. 8th ACM Symp. on Operating System Principles, 22-29.

Bernstein, P. A. and N. Goodman (1984) An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Trans. on Database Systems*, 596-615.

Birman, K. P., T. A. Joseph, T. Raeuchle, and A. E. Abbadi (1984) Implementing fault-tolerant distributed objects. 4th ACM Symposium on Reliability in Distributed Software and Database, 124-133.

Borg, A., J. Baumbach, and S. Glazer (1983) A message system supporting fault tolerance. 9th ACM Symposium on Operating System Principles, 90-99.

Dijkstra, E. W. (1965) Solution of a problem in concurrent programming control. *Communications of the ACM*, **8**(9), 569.

Dijkstra, E. W. (1968) Cooperating sequential processes. *In: Programming Languages*, pp. 43-112. F. Genuys, Academic Press, New York.

Fleisch, B. D. (Aug. 1986) Distributed system V IPC in locus: a design and implementation retrospective. Proceedings ACM SIGCOMM' 86 Symp. on Communications Architectures and Protocols, Stowe, Vermont, 386-396.

Khalidi, A., U. Ramachandran, M. Ahamad, and M. Yousef (1989) Coherence of distributed shared memory: unifying synchronization and data transfer. Proceedings 1989 International Conference on Parallel Processing, 160-169.

Leffler, S. J., R. S. Fabry, and W. N. Joy (Feb. 1983) A 4.2 BSD interprocess communication primer. Technical Report, Dept. of Computer Science and Electrical Engineering,University of California, Berkeley, CA, U.S.A.

Li, K. and R. Schaefer (1989) A hypercube shared virtual memory system. Proceedings 1989 International Conference on Parallel Processing, 125-132.

Popek, G., B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel (1981) A network transparent, high reliability distributed system. 8th ACM Symp. on Operating System Principles, 169-177.

Randell, B. (1975) System structure for software fault tolerance. *IEEE Trans. on Software Engineering*, **SE-1**, 220-232.

Schlichting, R. D. and F. B. Schneider (1983) Fail-stop processors: an approach to designing fault-tolerant distributed computing systems. *ACM Trans. on Computer Systems*, **1**(3), 222-238.

Schneider, F. B. (April 1986) Abstractions for fault tolerance in distributed systems. Technical Report TR86-745, Dept. of Computer Science, Cornell University, NY, U.S.A.

Walker, B., G. Popek, R. English, C. Kline, and G. Thiel (1983) The Locus distributed operating system. 9th ACM Symp. on Operating System Principles, 49-70.

Yap, K. S., P. Jalote, and S. Tripathi (1988) Fault tolerant remote procedure call. 11th IEEE Conf. on Distributed Computing Systems, 48-54.

一個分散信號機機制之設計與實作：DISEM

吳昭正　連秀娟　袁賢銘

交通大學資訊科學系

摘　要

對於解決並行程序間（concurrent process）的同步（synchronize）問題，信號機（semaphore）是一種優秀而且標準的機制；所以大多數作業系統都提供信號機機制讓使用者使用。本計畫主要在討論一個我們稱之爲DISEM的分散式信號機（distributed semaphore）之設計與實作。DISEM是被實作在執行具有BSD socket和System V IPCs的UNIX作業系統或其相關衍生版本之工作站的應用階層上，它能夠在分散式的工作站環境下，支援信號機機制。DISEN系統的使用方法是非常簡單容易的，它的使用方式就如平常使用者在程式中呼叫程式庫中的副程序一樣簡單，不必去考慮有關網路方面的問題。除此之外，它也對工作站的毀壞（crash）提供了可容錯的服務。從初步的效能評估結果中顯示出，DISEM系統的效能是可以接受的。整體而言，DISEM是一個在分散式的工作站網路環境下有用的且具有可攜性的信號機機制。