Design and Implementation of a Distributed Semaphore Facility*

Shyan-Ming Yuan, Chiao-Jang Wu, Hsiou-Mien Lien & I-Neng Chen
Department of Computer and Information Science
National Chiao Tung University
Hsinchu, Taiwan 30050

Abstract

This paper describes a distributed semaphore facility called DISEM. DISEM supports semaphore mechanism in a distributed workstations environment, it is implemented at the application level for workstations running a version or a derivative of UNIX operating system which supports BSD sockets and System V IPCs. In addition, DISEM also provides fault tolerant service in case a workstation crashes. Generally speaking, DISEM is a useful and portable facility for supporting of distributed semaphore in a local network of workstations.

1 Introduction

With progress in the area of computer network and operating systems, distributed computation has become more and more attractive in recent years. Since distributed computation involves the interaction of two or more processes, the distributed system must support some forms of synchronization and communication among them. Therefore, process synchronization is crucial for the design of the distributed system. Many problems involving distributed shared memory, atomic commitment and others require that a resource is allocated to a single process at a time. Conventionally, semaphores are the classic mechanism of solutions to these problems. Hence we design and implement a distributed semaphore facility to meet the requirement.

One of the principal advantages of distributed system is their potential for fault tolerance. 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 introduce a new architecture which best exploits this capability into the *Dis*tributed *Sem*aphore facility *DISEM*. The main advantages claimed for the new architecture are the reduction of the network traffic overhead due to fault tolerance and the relative simplicity of the strategy.

The remainder of the paper is organized as follows. Section 2 describes the definition of the distributed semaphore and reviews other corresponding work. Section 3 states the overall strategies of our design. Section 4 addresses the details of implementation and presents the user interface. Section 5 concludes the paper by emphasizing the important achievements of the project and our plans for future work.

*This work is supported in part by the Industrial Technology Research Institute, under contract number: I81004 and in part by the National Science Council, under contract number: NSC80-0408-E009-27.

2 Background and other work

This section gives some background for the semaphore and reviews some corresponding work about the distributed IPC.

2.1 Definition of distributed semaphore

Semaphore is a synchronization primitive that prevents two or more processes from accessing a shared resource simultaneously; our distributed semaphore is also the same. The processes can synchronize their operations on the different hosts by means of the distributed semaphore.

Dijkstra[1] published the Dekkers algorithm that describes an implementation of semaphore, integer-valued object that has the following atomic operations defined for it:

- Creation and initialization of a semaphore to a nonnegative value.
- A P operation that decreases the value of the semaphore. If the value of the semaphore is less than 0 after decreasing its value, the process that did the P goes to sleep.
- A V operation that increases the value of the semaphore. If the value of the semaphore becomes greater than or equal to 0 as a result, one process that had been sleeping as the result of a P operation wakes up.
- Close and remove an existing semaphore.

2.2 Other work

In distributed systems the implementation of mechanisms for communication and synchronization has traditionally taken one of two approaches: application-level based and OS (Operating System) kernel-level based.

The distributed System V IPC in Locus[2] is one of the early attempts at realizing a distributed semaphore mechanism. It is an OS kernel-level implementation and concerns itself with augmenting the Locus distributed operating system.

0-8186-2755-7/92 $03.00 © 1992 IEEE

However, the usefulness of such a system is limited because they were tailor-made for specific environments, required modification to the underlying kernel. The reality is that 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, UNIX will continue to be the preferred operating system by most users. Since new facilities for distributed and parallel processing are also being developed, requiring changes to the kernel in order to support a specific facility, becomes a costly and self defeating approach. The alternative is to develop these facilities entirely at the application level.

Consequently, DISEM is designed to require no changes to the UNIX operating system kernel. DISEM is implemented entirely at the application level. So it gains insight into the portability for all workstation running a version or a derivative of UNIX operating system which supports BSD sockets and System V IPCs.

3 Scenarios

In the following, we state the strategies used to implement our distributed semaphore facility.

3.1 Description of strategies

Our distributed semaphore is implemented in an abstract shared memory on a network. An abstract shared memory simulates a shared physical memory. Some location in the network implement the physical memory and all processes access the memory through the network operations. Therefore, a distributed semaphore looks as if a global shared resource in the distributed system. It is necessary that each of distributed semaphores has its own centralized arbiter for making control decisions to coordinate the requests from multiple clients. Usually, both the object and its arbiter of a distributed semaphore are located at the same site.

In our model, a DISEM plays two kinds of roles at the same time: one is the *arbiter* for some distributed semaphores, and another is the *agent* that is responsible to user for providing the distributed semaphore service. Each site involving the distributed semaphore mechanism runs the only one and the same copy of code for DISEM. Furthermore, in order to enhance the reliability of distributed semaphore and to reduce the bottleneck due to centralization of arbiters, we let these arbiters of distributed semaphores distribute fully in the network. In other words, all arbiters of distributed semaphores in the system are not centralized in DISEM of certain a site, but are distributed in DISEM of all sites which use distributed semaphores.

There is a problem of how to select a DISEM as the arbiter of a distributed semaphore? We choose the simplest and effective way: we let the DISEM of the site which first uses the distributed semaphore be the arbiter of that distributed semaphore. Therefore, performance is optimized because operations on the semaphore will be local operations for the arbiter.

3.2 Fault tolerant features

In a computing environment consisting of a local area network of workstations, a workstation may crash due to hardware or software failures. In order to make the distributed semaphore mechanism more reliable, some degree of fault tolerance is a mandatory requirement of such distributed facilities. Evidently, site failure of the arbiter of some distributed semaphores will lead to loss of all informations of some distributed semaphores so that these distributed semaphores could not work continuously. This is common folklore in distributed systems that a central server model is not a reliable way to construct a computer service.

Typically, the fault tolerance is enforced through replication. According to Yap, Jalote and Tripathi[3] and Bloch, Daniels and Spector[4] the distributed schemes (by contrast with non-distributed ones[5]) that use replication to support fault tolerance can be classified into three categories: the primary-standby approach[6], the modular redundancy approach[7] and the weighted voting approach[8]. Basically, our fault tolerance model which we call hot-standby strategy belongs to the modular redundancy approach. That is, each distributed semaphore has his own hot-standby replica at the other site. Other methods of achieving fault tolerance were considered too costly in terms of run-time delays. Our emphasis has been on simplicity as far as possible. Elegant but complex solutions are generally costly in terms of delay.

Usually, the hot-standby copy of a distributed semaphore is located at the site which is the second site uses this distributed semaphore in the distributed system. Therefore, it is fault tolerant provided that there exists at least one correct copy in the system except that the both sites crash simultaneously. Let us examine our model in following conditions.

First, we assume that failures are absent. In normal circumstances, a service request is sent to the primary replica of the distributed semaphore. Then, after the request is handled by DISEM of the primary replica, the request will propagate to the hot-standby replica of the distributed semaphore. The primary replica will wait for the acknowledgement of its hot-standby replica after its DISEM has sent out the request. Upon receiving that signal from its hot-standby replica, the primary replica sends the result to the requesting client. But from the viewpoint of client, the request is made only once, and only one copy of the result is received by the client. The request and the acknowledgement sent by the client and DISEMs are shown in Figure 1.

Secondly, the failure may occur in the primary replica. This kind of failure can occur at two times: before or after the DISEM of primary replica has propagated the request. If the failure occurs in the primary replica before the request is propagated to the standby, the client will detect that the primary replica had failed, and the same request will be sent to the hot-standby replica. Then everything continues just as if no error has occurred, except that the client resends the request and the result is sent directly from the standby to the client.

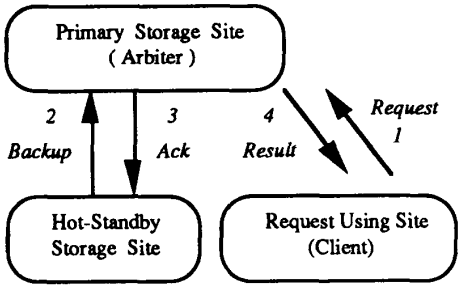

Figure 1: DISEM's operations

Now, let us consider the case of failure occurring in the primary replica after the request is propagated to its standby. If the primary replica fails after the request has propagated to the standby replica, the standby replica will receive the same request twice. Since the request carries with it a sequence number, the standby will process the request only once, and the result is sent directly from the standby to the client, rather than from the primary replica.

3.3 Recovery

Recovery from the failure is important for a fault tolerant system. Our recovery mechanism is rather simple compared with other complex schemes. Where the responsibilities of the failed primary will be taken over by its hot-standby, and then we must recreate a new standby replica to ensure that there are always available hot-standby replica. In our description, operations on distributed semaphores can involve up to two different sites. The using site is in the client machine that issues the request and receives the result; the storage site is the place where a distributed semaphore is stored actually.

Dangerous situation could occur if a process does a semaphore operation, presumably locking some resource, and then exits without resetting the semaphore value. Such situations can occur as the result of receipt of a signal that causes sudden termination of a process or site crash. Hence other processes would find the semaphore locked even though the process that had locked it no longer exists. To avoid such problems, it is necessary to back out the updates so that changes appear invisible. The mechanism which is used to implement this is an *undo log*. Each storage site records an undo log that indicates how to undo the semaphore if an application updates the semaphore and is aborted by a signal or site crash. Because semaphore does not survive site crashes, this undo log may be stored in memory. The undo log entries are removed when the semaphore is removed.

To avoid that the conditions described above occur, some nodes have to take part in recovery. Specifically, the nodes that are the storage sites for the failed semaphores and the storage sites whose distributed semaphores have been accessed by 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 system and executes the procedure $perform - recovery_i$ for the failed node. The procedure, shown in Figure 2, is executed by a node every time the failure of some node is detected.

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

Figure 2: Recovery procedure for a node i, on failure of an node N_k

The *resident_number* of semaphore S_j is the current number of storage site for semaphore S_j. The boolean array AVAILABLE is used to record the current status of a distributed semaphore. In part 1 of the recovery procedure, the fact that semaphore S_j has unavailed is recorded. This information is utilized by DISEM later. If a user requests distributed semaphore S_j that its AVAILABLE is False, DISEM will block the request until its AVAILABLE becomes to True.

Part 2 contains actions to be performed by the storage sites of distributed semaphore. If the node is the storage site of semaphore S_l that also stored in the failed node N_k, the node first becomes the primary storage site of the failed semaphore S_l. If there is the other using sites of semaphore S_l in system, the node will elect one of these using sites of semaphore S_l to act as the new hot-standby storage site of semaphore S_l. If the node is the storage site of semaphore S_l that have been accessed by the failed node N_k, then we use the information of *undo log* for semaphore S_l to undo the operations from the failed node N_k. Finally, the node will broadcast a success message to all nodes in system. After receiving this message for each node, its DISEM decreases the *resident_number* of semaphore S_l if it exists, by 1. Until the *resident_number* of semaphore S_l becomes zero, DISEM sets AVAILABLE of semaphore S_l to True and unblocks the request of semaphore S_l.

4 Implementation

We have implemented an initial version of the DISEM, running on a cluster of about ten Sun workstations connected by a 10Mbps Ethernet. This section outlines the implementation of DISEM.

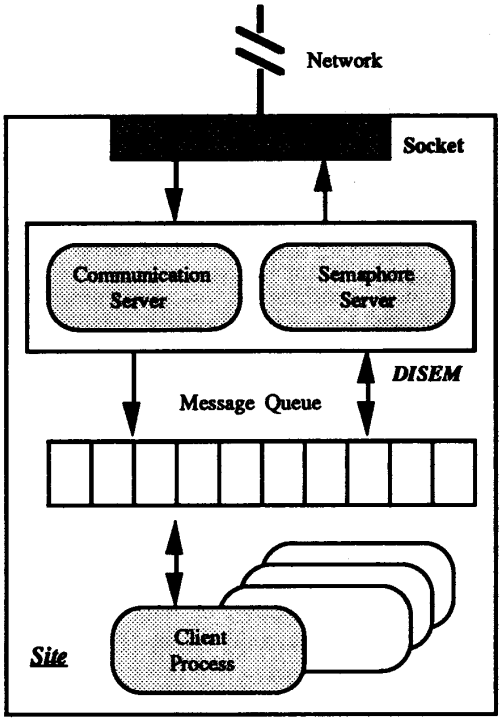

Figure 3: DISEM's architecture

4.1 DISEM's architecture

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 and for sending and receiving messages from all of the processes on its site. All intrasite communications are based on UNIX System V IPC — Message Queue. The communication server is responsible for receiving all messages from the socket on LAN, then relays these messages to the semaphore server through message queue.

All inter-site communication is based on SOCK_DGRAM protocol [9]. Figure 3 shows the two modules of DISEM.

4.2 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 network must be satisfied.

In UNIX System V, IPC uses two types of names: keys and handles[10]. A key is a 32-bit integer the user selects and associates with a message queue, a semaphore set, or a shared memory segment. Handles operate on objects. Typically when locates an object using a key, a handle 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 must provide a mechanism called ***using_table*** for locating 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. So there is no need for a separate name server in system and no need to worry about the reliability of name service.

In our DISEM system, the name *lookup* operation invoked by clients must return two things from 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 distributed semaphore being manipulated. In this case the token is typically an index into the distributed semaphore table of storage site; it saves the server from having to re-translate the name of distributed semaphore on each request.

Each entry in the using_table corresponds to one of the distributed semaphores being used in itself site: it contains the name of distributed semaphore, the address of the storage site, and a token. Initially, each client starts with an empty using_table. When a user requests to look up a distributed semaphore, the client 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 server then searches its local storage and one successful server (if any) responds to the request with the address of the storage site and the token for the distributed semaphore. Then, client uses the response to create a new entry in its using_table. Finally, the result of a lookup operation will return the handle (an index into its using_table) to the user. The handles serve the same purpose as the tokens described above: they allow the client 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 then it implicits that the distributed semaphore does not exist in the system and an error is returned to the user. Entries are added to the using_table only when needed: a distributed semaphore that has never been accessed in itself site will not appear in the using_table.

As stated above, a distributed semaphore's handle is returned as the result of a lookup operation. Application programmers typically do not inspect the handle. Rather, they present it to the DISEM to access an underlying object. Most function calls we will later describe take a handle as their first argument. As in UNIX System V, the handle is an integer and is an index's value. The index allows one to determine where, in a fixed size table, the pointer to the object is. When the object is removed, the reuse count in the index slot is increased so that the index's value is the offset into

the table times the reuse count. With this scheme, the possible range of handle values is increased to include all integers, not just small integers.

4.3 User interface

The usage of DISEM system is made entirely transparent. The DISEM enables the 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 users 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 semid=dsem_create(key_t key,
                      int init_val);
```

dsem_create() returns a positive value as the semaphore identifier (semid) or -1 when an error occurs.

If the distributed semaphore already exists, we do not initialize it and return -1. If the caller knows that the distributed semaphore must already exist. This dsem_open() function should be used, instead of dsem_create().

```
int semid=dsem_open(key_t key);
```

dsem_open() returns a positive integer as the semaphore identifier (semid) or -1 when an error occurs. If the distributed semaphore does not exist, we return -1.

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

```
int dsem_P(int semid, int amount);

int dsem_V(int semid, int amount);
```

The dsem_P() function decreases the distributed semaphore value by a user-specified amount. The dsem_V() function increases the distributed semaphore value by a user-specified amount. dsem_P() and dsem_V() return a value to indicate the operation is successful or failed.

The dsem_close() function is for a process to call before it exits, when it is done with the distributed semaphore. We decrease the count of the processes using the distributed semaphore in the systems, and if this is the last one, we can remove the distributed semaphore.

```
int dsem_close(int semid);
```

The dsem_close() also returns a value to indicate the operation is successful or failed.

5 Conclusion

This paper has outlined the objectives and significance of a distributed semaphore system — DISEM. We have overviewed issues related to DISEM and described the DISEM's implementation.

The fault tolerant distributed control strategy described in this paper is useful in situations for realizing fault-tolerant global control of resources in distributed computing systems. Fault tolerance is achieved by providing his own hot-standby replica at the other site for each distributed semaphore.

Future works include development of other distributed interprocess-communication facilities such as message queue and shared memory to form the package of distributed System V IPC. Also, future works are needed in improving the protection of access, management of replicas, examining the performance of a number of typical asynchronous concurrent programs using DISEM and the interworkstation communication efficiency.

References

1. E.W. Dijkstra, "Cooperating Sequential Processes," in *Programming Languages*, F. Genuys, pp. 43-112, Academic Press, New York 1968.

2. B. D. Fleisch, "Distributed System V IPC in Locus: A Design and Implementation Retrospective," *Proceedings ACM SIGCOMM' 86 Symp. on Communications Architectures and Protocols*, Stowe, Vermont, August 5-7, pp. 386-396, 1986.

3. K.S. Yap, P. Jalote & S. Tripathi, "Fault tolerant remote procedure call," *11th IEEE Conf. on Distributed Computing Systems*, pp. 48-54, 1988.

4. J.J. Bloch, D.S. Daniels & A.Z. Spector, "A weighted voting algorithm for replicated directories," *JACM, 34, (4)*, pp. 859-909, 1987.

5. J. Bartlett, "A NonStop kernel," *8th ACM Symp. on Operating System Principles*, pp. 22-29, 1981.

6. K.P. Birman, T.A. Joseph, T. Raeuchle & A.E. Abbadi, "Implementing fault-tolerant distributed objects," *4th ACM Symp. on Operating System Principles*, pp. 124-133, 1984.

7. J.S. Banino,J.C. Fabre,M. Guillemont,G. Morisset & M. Rozier, "Some fault-tolerant aspects of the chorus distributed system," *5th IEEE Conf. on Distributed Computing Systems*, pp. 430-437, 1985.

8. D.K. Gifford, "Weighted voting for replicated data," *7th ACM Symp. on Operating System Principles*, pp.150-162, 1979.

9. S.J. Leffler, R.S. Fabry, W.N. Joy, "A 4.2 BSD Interprocess Communication Primer," *Technical Report*, Dept. of Computer Science and Electrical Engineering, University of California, Berkeley, Feb. 1983.

10. M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, 1986.