Distributed Semaphores

Mahendra Ramachandran and Mukesh Singhal

Department of Computer and Information Science
The Ohio State University, Columbus, Ohio 43210-1277
email: {ramach,singhal}@cis.ohio-state.edu

Abstract

Semaphores provide a basic synchronization mechanism in uni- and multi-processor systems. Supporting semaphores in distributed systems has not received much attention. DSM systems provide a shared memory programming model on distributed systems and thus increase the appeal of providing efficient, decentralized solutions for the semaphore mechanism in a distributed system. We present a distributed semaphore mechanism in this paper. We use a two-level hierarchy to processors to increase locality of reference and thus the efficiency of access.

Key phrases: Operating Systems, Process Synchronization, Semaphores, Distributed Memory Architectures.

1 Introduction

Synchronization of concurrent processes has been an important issue in operating system design. Processes need to synchronize among one another or co-ordinate access to shared resources to maintain the correctness of the system. Semaphores provide a simple, yet powerful mechanism for synchronization among concurrent processes. In shared memory systems (uni-processor as well as multi-processor), a low-level hardware operation, Test&Set, is used to implement semaphores. This Test&Set instruction allows a process to read a memory address and modify it atomically. Various other implementations of semaphores for multi-processors have been proposed [1].

Implementation and properties of semaphores are well understood in shared memory systems. However, implementing semaphores in a distributed system has not received sufficient attention. A distributed system is a network of processors interconnected by a communication network. The processors do not share memory and exchange information through messages. The lack of shared memory makes implemention of semaphores very difficult in a distributed system. An implementation of semaphores in such a system must rely on message passing. To protect a critical section in a distributed system, several algorithms based on message passing have been proposed [2, 3, 4, 5, 6, 7]. Note that, these algorithms indirectly implement binary semaphore in distributed systems. There is still a need for general purpose resource counting semaphores in distributed systems. Providing such mechanisms through the use of a centralized controller is undesirable for several reasons. A single controller can be heavily swamped with requests, cause congestion of the network links near the controller node, it is the single point of failure of the system and the synchronization delay is twice the message delay.

In recent years, there has been wide-spread interest in implementing shared memory programming paradigm on distributed systems. Such systems, called Distributed Shared Memory (DSM), make it possible to execute shared memory parallel programs on a distributed system [8, 9, 10, 11, 12, 13].This makes it all the more important to provide shared variable-like synchronization mechanisms in distributed systems. Most DSM systems provide centralized implementations of mechanisms such as semaphores. A notable exception is Munin, which provides distributed locks for synchronization [14]. In this paper, we propose a distributed implementation of semaphores which efficiently performs **P** and **V** operations [15, 16] on semaphores.

The rest the paper is organized as follows. Section 2 describes the system model we use. The algorithm for accessing a semaphore is given in Section 3. The proof of correctness and performance of our algorithm is given in Sections 4 and 5 respectively. We conclude the paper in Section 6.

2 System Model and Definition

The distributed system is composed of a n of processors or nodes, labeled $P_0,...,P_{n-1}$. For the sake of clarity in describing our algorithm, we assume that each processor has only one process executing on it. Each processor queues up multiple requests for the same semaphore and services them sequentially.

The semaphore variables in the system are labeled $S_1,...,S_k$. Each semaphore variable has an initial value assigned to it (eg., $S_1 = 1$, $S_2 = 9$; S_1 is a binary semaphore and S_2 is a resource counting semaphore). In general, the initial value of $S_i = s_i$. To simplify the description, let us assume that s_i is a perfect square. Later we show that the model is easily extended when the value s_i is not a perfect square.

For a semaphore S_i with initial value s_i, we cluster the nodes in the system into $\sqrt{s_i}$ clusters. Each cluster contains $n/\sqrt{s_i}$ nodes. The first cluster contains the first $n/\sqrt{s_i}$ nodes and the second cluster contains the second $n/\sqrt{s_i}$ nodes and so on (So, $C_0 = \{P_0,...,P_{(n/\sqrt{s_i})-1}\}$ and $C_1 = P_{n/\sqrt{s_i}},...,P_{(2*n/\sqrt{s_i})-1}\}$, etc). For instance, if a system has 24 nodes and a semaphore S_i with an initial value of 16, Figure 1 shows the four clusters with $24/\sqrt{16} = 6$ nodes per cluster.

A copy of the semaphore S_i is maintained in each cluster of the system. However, the initial value of the local copy is $\sqrt{s_i}$ and not s_i. Each node has two request sets defined, the local request set and the global request set. The local request set (LR) of a node contains all the nodes in its cluster. The global request set (GR) contains one node from each of the $\sqrt{s_i}$ clusters. Formally we define the requests sets as:

$$LR_i = \{P_I | P_I \textit{ is in the same cluster as } P_i\}$$

$$GR_i = \{P_I | I = i \pm k * n/\sqrt{s_i}, k = 0, ..., \sqrt{s_i}\}$$

If a semaphore S_i's initial value of s_i is not a perfect square, we divide the system

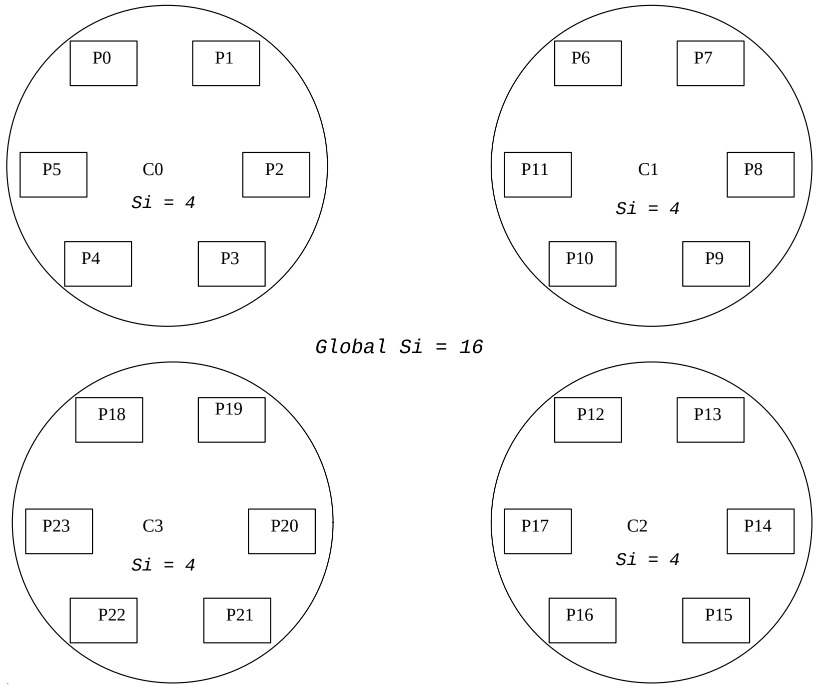

Figure 1: Clustering the Processors for semaphore S_i

into $\lfloor\sqrt{s_i}\rfloor$ clusters, each with $n/\lfloor\sqrt{s_i}\rfloor$ nodes. If each cluster's copy of the semaphore is initialized to $\lfloor\sqrt{s_i}\rfloor$, then the global value of the semaphore would be $\lfloor\sqrt{s_i}\rfloor^2$. So, we must distribute the remaining value $s_i - \lfloor\sqrt{s_i}\rfloor^2$ among the clusters. Hence, the local copy of the semaphore is now set to $\lfloor\sqrt{s_i}\rfloor + (s_i - \lfloor\sqrt{s_i}\rfloor^2)/\lfloor\sqrt{s_i}\rfloor$. An Alternative solution is to divide the network into $\lceil\sqrt{s_i}\rceil$ clusters. Then $\lfloor\sqrt{s_i}\rfloor$ of the clusters will have access to semaphores initialized to $\lfloor\sqrt{s_i}\rfloor$ and the last cluster will have its semaphore initialized to $(s_i - \lfloor\sqrt{s_i}\rfloor^2)$.

Similarly, if the number of nodes in the system does not divide $\sqrt{s_i}$, we assign $\lfloor n/\sqrt{s_i}\rfloor$ nodes to each of the $\sqrt{s_i}$ clusters and the remaining $n - \sqrt{s_i}\lfloor n/\sqrt{s_i}\rfloor$ nodes among the $\sqrt{s_i}$ clusters, one per cluster. This would result in a difference of at most one node between any two clusters.

Lastly, operations defined on a semaphore are the standard **P** and **V** from traditional operating systems:

```
P(Sem):      if (Sem >=  1)
               then   Sem := Sem - 1;
               else   block the process on the semaphore queue;

V(Sem):      if (semaphore queue is not empty)
                then  de-queue a process;
                else  Sem := Sem + 1;
```

3 Accessing A Semaphore

When a process wishes to perform a **P** operation on a semaphore, it is important that the **P** operation be performed in a mutually exclusive manner to maintain correctness of the semaphore. In shared memory multiprocessors, this **P** operation can be performed mutually exclusively through the use of test&set instructions on a shared memory location. In a distributed system with no central controller to maintain the semaphore, it is necessary for the nodes of the system to take extra measures to ensure that the **P** operation is performed in mutual exclusion. Likewise, a **V** operation should also be performed in mutual exclusion.

In the previous section, we described how the nodes in the system are divided into clusters. This was done to localize message traffic on the network to nodes within the cluster when accessing semaphores. Furthermore, each cluster's copy of the semaphore which is initialized to $\sqrt{s_i}$, is mapped onto any node in a cluster. When a node needs to access a semaphore, it is migrated to that node. We next describe how a node acquires a semaphore to perform **P** and **V** operations. We first describe the manner in which a node acquires a semaphore from within its cluster and then if necessary from another cluster in the system.

3.1 Intra-Cluster Access

P operation

In order to perform a **P** operation on a semaphore S_i, a node P_j must first acquire the semaphore. If the node already holds the semaphore, it performs the $\mathbf{P}(S_i)$ without generating any network traffic. But, if it does not hold the semaphore, the node must acquire it from the node holding of the semaphore in its cluster since only the holder of the semaphore can perform a **P** operation on it.

Acquiring a semaphore implies that the node which currently holds the semaphore must be determined. This can be accomplished in several ways. One approach is to broadcast the request message to every node in the cluster [5]. An alternative approach is to maintain a probable_holder variable on each node that indicates the node likely to hold the semaphore [6]. This variable at a node indicates the node to whom it passed the semaphore most recently. The request message is sent to the node specified by the probable_holder variable. That node in turn might have to forward the message until it arrives at the current holder of the semaphore. Both the ways have their trade-offs. The broadcast approach can acquire the semaphore with two message delays while the probable_holder method would involve on average $O(log\sqrt{N}) + 1$ message delays. The broadcast approach, however, results in excessive message traffic ($|LR| -1$ messages per request) on the network. A reasonable compromise might be to maintain a history of recent holders of the semaphore and multicast the request message to a subset of the nodes in the LR set [7]. We are conducting simulation studies of these approaches to determine their suitability for various traffic patterns. In the rest of this section we assume the probable_holder method to access the semaphore.

When the requesting node becomes the holder of the semaphore, it performs the **P** operation. If the semaphore's value is greater than zero, the **P** operation is completed. Otherwise, the node must wait for $\mathbf{V}(S_i)$ to be performed by another node before it can complete the **P** operation. The code to perform a **P** operation on a the semaphore is shown in Figure 2. The symbols $\ll$ and $\gg$ in the code denote a per-node critical section.

```
if (semaphore is not available) then
    send P_request message for Sem_i to probable_holder node.
    wait for the semaphore.
endif;

<< if (Sem_i > 0) then
     Sem_i := Sem_i - 1; >>
   else
     block process.
```

Figure 2: Executed by a node to access a semaphore to perform a **P** operation.

If a *P_request* message arrives at a node that currently holds the semaphore, the node sends the semaphore to the requesting site provided it is not using the semaphore. If it is performing the **P** operation or is blocked, it will queue up the request. Upon completion of the **P** operation, the semaphore is sent to the requesting site. If there is more than one request queued, the semaphore along with the rest of the requests are sent to the request at the head of the queue. In this manner blocked requests are forwarded along with the semaphore and eventually when a request gets to the head of the queue, the node which had initiated that request becomes the next recipient of the semaphore.

The semaphore is forwarded to the next requesting node regardless of whether the holder node has as yet to perform a **V** operation on the semaphore or whether the semaphore's value is zero. It is tempting to suggest an improvement wherein the request is blocked until the semaphore's value is non-zero, as a result of a local or remote **V** operation on the semaphore. However, we do not propose it because this may conflict with the inter-cluster access we describe in the following section.

If a node that receives a request no longer holds the semaphore, it forwards the message to the probable holder as specified by its local variable. However, if it is in the process of acquiring the semaphore, it blocks the request until it acquires the semaphore and finishes its **P** operation. The code in Figure 3 shows this process.

```
<< On receipt of a P_request message for Sem_i.
   if (Performing a P operation on the Sem_i) then
         queue the P_request message.
   else if (Sem_i is locally available) then
         SEND Sem_i to Node_j;
         probable_holder = Node_j;
   else
        SEND P_request to probable_holder;
 >>
```

Figure 3: Executed by a node on receipt of a *P_request* message.

V operation

A node that wishes to perform $\mathbf{V}(S_i)$, may not find the semaphore available locally. If the semaphore is available locally, the node performs the $\mathbf{V}(S_i)$ immediately. Otherwise, rather than acquiring the semaphore to perform the **V**, it sends a *V_request* message to the node currently holding the semaphore. This is done by sending the message to the probable_holder node just as was done by the node to acquire the semaphore. The message may need to be forwarded until it arrives at the current holder of the semaphore which then performs the **V**. The request can be viewed as a remote or proxy **V** operation that is performed on behalf of the requesting node. Figures 4 and 5 show the code executed by the nodes in initiating and servicing *V_request* messages.

```
<< if (Sem_i is available locally) then
        Sem_i := Sem_i + 1;
   else
        SEND  V_request message for Sem_i to probable_holder;  >>
```

Figure 4: Executed by a node to perform a **V** operation.

```
On receipt of a V_request from Node_j for Sem_i
<< if (Sem_i is available locally) then
      if (blocked for Sem_i) then
          unblock process
      else
          Sem_i := Sem_i + 1;
    else
      V_request message for Sem_i to probable_holder;  >>
```

Figure 5: Executed by a node on receiving of a *V_request* message.

3.2 Inter-Cluster Access

If only intra-cluster access is used to implement **P** and **V** operations, the following scenario is possible. Suppose node P_i in cluster C_j acquires the cluster copy of semaphore S_k and finds the value to be zero and blocks for a $\mathbf{V}(S_k)$ operation to be performed. In this situation, if every other node in the cluster issues a **P** operation before doing a **V** operation, then the P_i will be forced to wait indefinitely even though the value of S_k in the whole system may be greater than zero. By acquiring the semaphore from another cluster in the system, such situations can be avoided.

Another motivation for inter-cluster access is efficiency. A node which cannot complete the **P** operation on a semaphore because the value is zero, need not wait for one of the other nodes in its cluster to execute a **V**. Instead, it should be able to access the copy of the semaphore from another cluster, provided one exists with a non-zero value, expediting the execution of the **P** operation.

In order to perform **P** and **V** operations on a copy of the semaphore in another cluster, a node makes use of the nodes in its *GR* set as proxy nodes in each of the other clusters in the network. When a node is unable to complete the **P** operation on a local semaphore (because its value is zero), it sends a *P_proxyrequest* message to one of the nodes in its *GR*. It waits for a response before trying the next node in *GR*.

A node which acts as a proxy on behalf of the requesting node accesses the semaphore in its cluster in the same manner as described in the previous section. If it finds the semaphore is zero, it responds to the *P_proxyrequest* message with a *deny* message (it does not initiate another inter-cluster access). If it succeeds in performing the **P** on the semaphore, it responds with a *success* message to the requesting node. If the proxy node was not successful in performing a **P** operation on its behalf, the requesting node sends a request message to the next node in the *GR* set.

To perform a proxy **V** operation on the semaphore, the node sends a *V_request* message to the node which performed the proxy **P** operation. There is no difference between the request message used here and the one used for intra-cluster remote **V** operation because the node receiving the request treats both messages in the same manner.

Rather than have a proxy node perform a **P** and a **V** on behalf of a requesting node, we could treat the inter-cluster access as a transfer of a semaphore value from one cluster to the other. Thus, after a proxy **P** operation, the requesting node will not send a *V_request* message to its proxy node. Instead, it will initiate a $\mathbf{V}(S_k)$ operation on its cluster's copy of the semaphore. This results in the local semaphore effectively gaining a value. This migration of semaphore values results in a dynamic distribution/sharing of semaphores among clusters where higher accesses to semaphores is observed. Even in applications with homogeneous access patterns to semaphores, there could be statistical fluctuations which make such dynamic sharing attractive. In applications with hetrogenous access patterns, the effectiveness of this scheme should be even more pronounced. We intend on carrying out performance studies based on simulations of various access patterns.

While awaiting a response to a proxy request, it is possible that a requesting node receives a *V_request* message. This message may either be an intra-cluster **V** request or an inter-cluster proxy-**V** request. By servicing such a request, it would be possible for the blocked node to complete its **P** operation locally. It is important for the requesting node to periodically check for such a message even while trying to perform a **P** through a proxy node. If it receives a *deny* message from a proxy node, it should first check for the presence of a *V_request* message before sending a *P_proxyrequest* to another proxy node. On the other hand, if the requesting node receives both a *success* and a *V_request* message, it would proceed with the completion of the **P** operation based on the *success* message. After completion of the **P**, it would process the *V_request* message and if necessary forward the semaphore to another node in its cluster.

The modified algorithms to perform **P** operations and servicing remote requests are presented in Figures 6 and 7, respectively. The algorithms in Figures 4 and 5 for initiating a **V** operation and for servicing a **V** request are essentially similar and not reproduced. Servicing *P_requests* requests is still the same as the code in Figure 3.

```
if (semaphore is not available) then
    send P_request message for Sem_i to probable_holder node.
    wait for the semaphore.
endif
<< if (Sem_i > 0) then
      Sem_i := Sem_i - 1;    >>
   else
      While (NOT done) do
          SEND P_proxyrequest message for Sem_i to a node in GR.
          AWAIT reply.

          << if (reply = ''success'') then
               done := TRUE;
            else if (V_request message has arrived) then
                   done := TRUE;
                   Execute code described in Figure 5.
                 endif  >>
      endwhile
```

Figure 6: Executed by a node to initiate a **P** operation (both intra- and inter-cluster).

```
<< On receipt of a P_proxyrequest message for Sem_i from Node_j
   if (Sem_i NOT locally available) then
       SEND P_request to probable_holder;
       wait for semaphore;
   else if (requesting the semaphore myself) then
       queue the P_proxyrequest message until semaphore is available.>>

<< if (Sem_i > 0 ) then
       Sem_i := Sem_i - 1;
       SEND ''success'' to Node_j;
   else
       SEND ''deny'' to Node_j;    >>
```

Figure 7: Executed by a proxy node on receiving a *P_proxyrequest* message.

3.3 Combining Intra-Cluster and Inter-Cluster Operations

In the previous section, an inter-cluster operation is performed only after an intra-cluster access to the semaphore is accomplished, and the semaphore's value was zero. This increases the delay in performing a **P** operation because of serial execution of the intra-cluster and inter-cluster access. This delay can be reduced by combining the intra-cluster and the inter-cluster operations into one phase. However, it will increase the number of messages required to perform **P** operations. While this increase in the number of messages may be tolerable under light semaphore access patterns, it may degrade the performance under heavy loads. This combined access may still be beneficial in cases where the semaphore within the cluster is typically at or close to zero.

The combined access strategy works as follows. When a node performs a **P** operation, it broadcasts the *P_request* message to all the nodes in its *LR* set as well as a *P_proxyrequest* message to a node in its *GR* set. The nodes in the *GR* act as proxy nodes for the node making the request. The proxy node acquires the semaphore from within its cluster and if possible performs the $\mathbf{P}(S_i)$. It then replies to the requesting node with either a *success* or *deny* message to indicate whether or not it succeeded.

The requesting node may either receive the semaphore followed by the reply from the proxy node, or the reply followed by the semaphore, or it could receive both simultaneously. We consider the three cases separately below.

If it receives the semaphore first and is able to perform the **P** operation on it, the node must ensure that the subsequent arrival of a proxy *success* message is responded to with a *V_request* message. If the proxy message is a *deny*, it can be discarded. If it was unable to perform the **P** operation on the local semaphore, it must await the result of the proxy request. If the reply is a *deny*, it issues proxy requests to the other nodes in its *GR* set as was done in the previous section.

If the requesting node receives the proxy node's response before the semaphore, it determines whether the proxy node was successful in performing the proxy **P** request. If it was, then the requesting node's **P** operation is complete. If not, the requesting node issues a new proxy request to another node in its *GR* set.

Lastly, if the response and the semaphore arrive simultaneously, the proxy node's response is first examined. If the message is a *success*, the **P** operation is complete and the semaphore is not examined. If the message is a *deny*, the semaphore is examined and if possible, the **P** operation is performed on it. If the semaphore is already zero, then a new proxy request is sent to another node in the *GR* set.

4 Correctness

We now present a correctness proof for our algorithm. In order to do so, we first prove that **P** and **V** operations are done in a mutually exclusive manner. To prove deadlock freedom, we need to show that no set of nodes form a cycle while trying to acquire the semaphore to perform a **P** operation. Lastly, we show that any process wishing to perform a **P** operation will do so in finite time (i.e., starvation freedom).

In this section we denote the probable_holder field of a node P_i as $\mathcal{PH}(P_i)$ for brevity. For example, $\mathcal{PH}(P_i) = P_j \Rightarrow P_i$'s probable_holder field indicates that P_j is the probable holder of the semaphore.

4.1 Mutually Exclusive P and V Operations

Observation 1 *Only one copy of the semaphore exists in a cluster. There may be $\sqrt{s_i}$ copies in the system, but a semaphore copy never leaves its cluster.*

Theorem 1 *The* **P** *and* **V** *operations are done in a mutually exclusive manner.*

Proof: There is only one copy of the semaphore per cluster and only the node holding it can perform **P** and **V** operations on it. Mutual exclusion is guaranteed by the node by performing the operations within a critical section as shown in our code. □

4.2 Deadlock Freedom

To prove the absence of deadlock in our algorithm, we must show that two or more nodes will not form a cycle in trying to acquire a semaphore from each other. We also show that a *P_request* by a node will not loop back to it and cause it to self-deadlock. We first state a couple of observations which simplify our proof.

Observation 2 *Initially, $\mathcal{PH}(P_i) = P_j, \forall\ P_i \in C_j$ and P_j is the holder of the semaphore in cluster C_j.*

Observation 3 *$\mathcal{PH}(P_i) = P_i$ implies that P_i holds the semaphore at present and is at the root of the tree that spans the nodes of its cluster.*

The above observations state that the probable_holder fields induce a spanning tree within each cluster. The initial tree is like a star with the root or hub holding the semaphore and all the other nodes being leaves of the tree. As requests are made and the semaphore is propagated, the root of the tree changes in concert with the propagation.

Lemma **1** *A node's request for a* **P** *or* **V** *operation will not form a cycle.*

Proof: P_i sends its request to P_j provided $\mathcal{PH}(P_i) = P_j$. This traversal of the request is denoted by $P_i \rightarrow P_j$. For a cycle to result from the forwarding of the request message, the following path must be traversed by the request: $P_i \rightarrow P_j \rightarrow P_k \rightarrow \cdots P_q \rightarrow P_m \rightarrow P_i$.

A node such as P_m cannot exist along the path traversed by the request. For P_m to be on the path taken by the request message would imply that P_m was on the path along which the semaphore propagated after leaving P_i (the last time). But, then P_m must either still hold the semaphore or passed it to a node other than P_i.$\Box$

Lemma **2** *Two or more nodes will not form a cycle while waiting for a semaphore.*

proof: We can assume that none of the nodes in question hold the semaphore, since a node which currently holds a semaphore will not make a request for it. Thus, if there exists a cycle, it must be a knot since none of the nodes involved is a sink.

For two nodes, P_i and P_j to form a cycle while trying to access a semaphore, $\mathcal{PH}(P_i) = P_j$ and $\mathcal{PH}(P_j) = P_i$. But, $\mathcal{PH}(P_i) = P_j \Rightarrow P_j$ received the semaphore from P_i. Similarly, $\mathcal{PH}(P_j) = P_i \Rightarrow P_i$ received the semaphore from P_j. Furthermore, as stated in observation 3, it is impossible for either P_i or P_j to maintain an old value for $\mathcal{PH}$. Thus it is impossible for both $\mathcal{PH}(P_i) = P_j$ and $\mathcal{PH}(P_j) = P_i$ to hold.

For three nodes, P_i, P_j and P_k to form a cycle, we must have each node waiting on one another. For instance, $P_i \rightarrow P_j \rightarrow P_k \rightarrow P_i$. For this to occur, $\mathcal{PH}(P_i) = P_j$, $\mathcal{PH}(P_j) = P_k$ and $\mathcal{PH}(P_k) = P_i$ must hold. This implies that P_i held the semaphore prior to

P_j and P_j held it before P_k. By transitivity, P_i must then have held it before P_k, which contradicts the statement $\mathcal{PH}(P_k) = P_i$. A similar argument can be used for more nodes as well. □

Theorem 2 *The algorithm is deadlock free.*

Proof: The preceding lemmas have proven that a node will not block its own request, that two or more nodes will not block on each other and form a knot. Thus, the algorithm is deadlock free. □

4.3 Starvation Freedom

Lemma 3 *Every* P_request *message eventually reaches the semaphore holder.*

Proof: Suppose P_i issues a *P_request* message. This message is sent to the node specified by $\mathcal{PH}(P_i)$. The node specified by $\mathcal{PH}(P_i)$ is the one to which P_i sent the semaphore last. If node $\mathcal{PH}(P_i)$ holds the semaphore, then the request has reached the holder. If the semaphore is no longer at node $\mathcal{PH}(P_i)$, then the message is forwarded to the node $\mathcal{PH}(\mathcal{PH}(P_i))$. This third node similarly is the one to which node $\mathcal{PH}(P_i)$ sent the semaphore last. By recursively constructing this argument, we see that the $\mathcal{PH}$ of each node defines a directed edge in a path from P_i to the current holder of the semaphore. □

Lemma 4 *A request message will not be queued indefinitely.*

proof: *V_request* messages are processed as soon as the current holder of the semaphore can do so and are thus not queued in the same manner as *P_request* messages.

A *P_request* message by a node P_i may be queued at the holder of the semaphore or at an intermediate node which is itself waiting for the semaphore.

Suppose P_i is queued up at the holder of the semaphore and there are k nodes ahead of it on the queue. Note that $k \leq n/\sqrt{s_i} - 2$. After the semaphore along with the queue is forwarded to each of the k nodes blocked ahead of it in the queue, P_i will receive the semaphore and the queue.

Suppose P_i is queued up at an intermediate node, P_j. Assuming P_j's request is queued up at the semaphore holder, then after a delay of at most $n/\sqrt{s_i} - 3$ nodes, P_j will receive the semaphore. If P_i is the k^{th} node on the queue at P_j and if the queue sent with the semaphore is of length k', then as per the algorithm, the queue at P_j will be appended to the queue it received. Thus after the node P_j has received the semaphore, P_i will be the $(k' + k)^{th}$ node to receive the semaphore. Similarly, if P_j itself was queued at the intermidiate node, by using a similar argument, we can see that after at most $n/\sqrt{s_i} - 2$ nodes, each with a local queue of blocked request of at most $n/\sqrt{s_i} - 3$, P_i will get the semaphore (though the actual number of requests ahead of it will be bounded by $n/\sqrt{s_i} - 2$). □

Lemma 5 *A Proxy request message will not be queued indefinitely.*

proof: A proxy request is sent by a node P_i to a node P_{prj} from its GR set. P_{prj} will then issue a intra-cluster request for the semaphore which, as proven in the above lemma, will not be delayed indefinitely. Upon checking the state of the semaphore, P_{prj} and respond with a success or a deny message. Thus, P_{prj} will respond to P_i's request in a finite time. □

Theorem 3 *Every request for a* **P** *or a* **V** *operation on semaphore is eventually satisfied.*

Proof: It has been proven that every request eventually reaches the semaphore holder and that a request will not be queued up indefinitely in the preceding lemmas. □

5 Performance

We are interested in how our algorithm performs in terms of the number of messages required to perform **P** and **V** operations. We are also interested in determining the delay involved in these operations. There are two ways of measuring these: analysis and simulation. Since analytical modeling of these algorithms is extremely difficult for the general case, we simulate the behavior of our system. We are currently in the process of simulating our system for a wide range of semaphore request patterns. (We will present our results in detail in a subsequent technical report.)

In considering the average case complexity of performing **P** and **V** operations, there are two cases to be considered: intra-cluster and inter-cluster. If a node has to acquire a semaphore (or send a *V_request* for a **V** operation), in the best case the semaphore is held by the node which is specified by the probable_holder field. In the worst case, however, the request message must be forwarded to every node in the cluster. This would require $n/\sqrt{s_i} - 1$ messages to get to the holder node. As was shown in [6], on average we expect $O(log(n/\sqrt{s_i}))$ messages.

If the holder node does not have any other requests pending, the current request can be satisfied immediately, incurring one additional message. However, if there are blocked requests queued at the holder, then the current request would be added to that queue and be serviced when it comes to the head of the queue. If there are Q requests ahead of it, then the total delay in accessing the semaphore would be $Q * (average\ P\&V\ execution\ delay\ +\ message\ delay)\ +\ 1\ message\ delay$

It is worth noting that a request may be blocked at a non-holder node, which is itself blocked either at the holder or yet another node, and so on. The above equation would still apply and $Q \leq n/\sqrt{s_i}$. If an inter-cluster proxy operation is necessary, this would require an additional round-trip message delay plus the intra-cluster delay.

6 Conclusion

Semaphores are a basic, yet powerful synchronization mechanism in uni-processor and multi-processor systems. In light of recent advances in distributed shared memory programming models, it is particularly appealing to provide the semaphore for synchronization in distributed systems. In this paper, we presented a scheme for performing operations on semaphores in a distributed environment.

We group the nodes in a distributed system into a two-level hierarchy called clusters. The sizes of the clusters depends on a semaphore's value. For the sake of reducing the amount of information that needs to be maintained, the system can define a few cluster sizes (instead of one for each semaphore) and associate the most appropriate one for each semaphore. This system dependence makes the concept of dynamically migrating semaphore values from one cluster to another to achieve access-load sharing all the more appealing.

Simulations of the scheme is under way and we will present a detailed analysis in a subsequent paper.

References

[1] Anne Dinning. "A Survey of Synchronization Methods for Parallel Computers". *IEEE Computer*, 22(7):66–77, July 1989.

[2] Leslie Lamport. Time, Clocks and Ordering of Events in Distributed Systems. *Communications of the ACM*, July 1978.

[3] G. Ricart and A. K. Agrawala. An Optimal Algorithm for Mutual Exclusion in Computer Networks. *Communications of the ACM*, Jan 1981.

[4] M. Maekawa. A $\sqrt{N}$ Algorithm for Mutual Exclusion in Decentralized Systems. *ACM Transactions on Computer Systems*, May 1985.

[5] I. Suzuki and T. Kasami. A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, Nov 1985.

[6] M. Naimi and M. Trehel. An Improvement of the log(n) Distributed Mutual Exclusion Algorithm. In *Proc. of the 7th International Conference on Distributed Computing Systems*, 1987.

[7] Mukesh Singhal. A Heuristically-Aided Algorithm for Mutual Exclusion in Distributed Systems. *IEEE Transactions on Computers*, 38(5), May 1989.

[8] Kai Li. *"Shared Virtual Memory on Loosely Coupled Multiprocessors"*. PhD thesis, Yale University, September 1986.

[9] John Bennett, John Carter, and Willy Zwaenepoel. "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence". In *Symp. on Principles and Practice of Parallel Programming*, pages 168–175, March 1990.

[10] U. Ramachandran and M. Y. A. Khalidi. An Implementation of Distributed Shared Memory. *Software - Practice and Experience*, 21(5):443–464, May 1991.

[11] Roberto Bisiani and Mosur Ravishankar. "Plus: A Distributed Shared-Memory System". In *International Symp. on Computer Architectures*, pages 115–124, May 1990.

[12] Brett Fleisch and Gerald Popek. "Mirage: A Coherent Distributed Shared Memory Design". In *Proceedings of the eleventh ACM Symp. on Operating System Principles*, pages 211–223, December 1989.

[13] Bill Nitzberg and Virginia Lo. "Distributed Shared Memory: A Survey of Issues and Algorithms". *IEEE Computer*, pages 52–60, August 1991.

[14] John Carter, John Bennett, and Willy Zwaenepoel. "Implementation and Performance of Munin". In *ACM Symp. on Operating System Principles*, pages 152–164, October 1991.

[15] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, page 569, Sep 1965.

[16] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. Academic Press, London, 1968.