Migrating Controller Based Framework for Mutual Exclusion in Distributed Systems

Satyendra Rana[1], Krishna Raman[1] and Vipin Chaudhary[2]
Wayne State University
Detroit, MI 48202

Abstract

A new framework for migrating controller based distributed mutual exclusion algorithms is presented. A salient feature of the proposed framework is the separation of two orthogonal aspects of the problem, viz., migrating controller and the granting of critical section entry. Two new algorithms are presented to illustrate the derivation of specific algorithms from the generalized framework.

Keywords : *Mutual Exclusion, Distributed Systems, Generalized Frameworks, Derivation of algorithms*

1 Introduction

The mutual exclusion problem has received a lot of attention for its significance both as an important synchronization problem and as a paradigm for studying distributed decision-making algorithms. The research efforts for the above problem can be broadly divided into two categories: 1) proposing algorithms with some improved performance measure, and 2) devising generalized frameworks from which a spectrum of algorithms can be generated. This paper belongs to the second category.

In this paper, we present a new framework for **migrating controller** based algorithms. Our framework subsumes many existing algorithms and is comparatively simpler in exposition. Furthermore, it opens up directions for deriving new, efficient, and adaptive algorithms. A salient feature of our framework is the separation of two orthogonal aspects of the problem, viz., migration of controller and the granting of *critical section* entry. Because of this separation, controller migration frequency can be reduced and thus, the overhead is distributed among several critical section entries, thereby reducing the average complexity.

2 Single Controller Based Algorithms

We are given a set of n processes, P_i, $1 \leq i \leq n$, which communicate among themselves by message passing. The processes do not share a global memory. The message passing is asynchronous and reliable with finite delay. Each process has a critical section code. The distributed mutual exclusion problem is to derive a protocol to be followed by processes before entry and after exit of their critical section codes such that at any instant at most one process is in its critical section code, and further any process trying to enter the critical section must succeed in doing so in a finite time.

We consider a class of algorithms where a unique process is designated as the controller and has the responsibility for scheduling critical section requests. When the controller responsibility is dynamically delegated among processes, we get **migrating controller** based algorithms.

2.1 Architecture

We associate a unique process M_i to each process P_i and refer to M_i as a *mutual exclusion server* (MUTEX server or simply server) and P_i as the client of M_i. The term *node* i refers to both P_i and M_i collectively. It is the MUTEX servers that interact with each other to implement a distributed mutual exclusion algorithm. The MUTEX servers are assumed to be fully connected and have a peer to peer relationship.

A message-based interface is provided between the client and the corresponding MUTEX server. Message identifiers with boldface refer to messages exchanged between a client and MUTEX server, whereas message identifiers in slant refer to those exchanged among MUTEX servers.

Each MUTEX server always performs a non-controller role and also has the capability to be the controller. However, at any instant, at most one MUTEX server is the controller. Migration of the controller involves a server ceasing to be in the controller role and another server assuming the controller role.

2.2 A Generalized Framework

In the generalized framework, we employ two abstract distributed information types, referred as REQUESTS_INFO and MIGRATION_INFO, and the corresponding operations defined on these. Particular algorithms can be easily derived by implementing REQUESTS_INFO and MIGRATION_INFO.

2.2.1 The abstract type REQUESTS_INFO

The REQUESTS_INFO type encapsulates all information necessary for routing and scheduling requests for critical section

[1] Authors are with Parallel and Distributed Systems Laboratory, Department of Computer Science.

[2] This author is with Parallel and Distributed Computing Laboratory, Department of Electrical and Computer Engineering. His research was supported in part by NSF Grant MIP-9309489 and Wayne State University Faculty Research Award.

0-7803-2492-7/95 $4.00 © 1995 IEEE

entry. This information includes the collection of pending requests and a *cs_status*. The *cs_status* is either **free** (no pending requests and no process is in critical section) or **busy**. For now, we are merely concerned with the operations exported by REQUESTS_INFO. These operations are:

- *register_request*(): This operation is invoked to insert the request of a node for critical section entry in the REQUESTS_INFO object. MUTEX servers invoke this operation in the non-controller role. The *register_request* operation requires the *id* of the requestor. Additional arguments may be needed depending on the implementation of REQUESTS_INFO.
  Correct implementation of *register_request* must ensure the following properties:
  - **Property 1** *Within a finite time after invocation of register_request(i), i is inserted into the collection of pending requests.*
  - **Property 2** *If cs_status =* **free** *when i is inserted in the collection, a message is sent to notify the current controller.*
- *select_request*(): The operation *select_request* is invoked only in the controller role. It returns the *id* of a process which is to be granted the critical section entry next. Correct implementation of *select_request* must ensure the following property:
  - **Property 3** *The operation select_request() completes within finite time. If there are no pending requests, it sets cs_status* **free** *and returns* **null**. *However if there are pending requests in the collection, select_request() sets cs_status* **busy** *and returns a request by choosing one from the collection using a starvation free scheduling criterion. The request is also removed from the queue.*
- *update_controller_info*(): The operation *update_controller_info* is also invoked in the controller role. It is invoked when the current controller relinquishes the controller role to another server. The correctness properties to be ensured by this are:
  - **Property 4** *The operation update_controller_info() completes within finite time.*
  - **Property 5** *The concurrent execution of update_controller_info() and register_request() does not violate properties 1, 2 and 4.*

2.2.2 The abstract type MIGRATION_INFO

In an instance of type MIGRATION_INFO, the information needed for migration is maintained. The operations exported by MIGRATION_INFO are:

- *test_migration_condition*(): The operation *test_migration_condition*() returns **true** if the migration must take place at this instance otherwise it returns **false**. The correctness requirement is:
  - **Property 6** *The operation test_migration_condition() completes within finite time.*
- *select_next_controller*(): The operation *select_next_controller*() returns the identity of the MUTEX server selected to assume the controller role next. The correctness requirement is:
  - **Property 7** *The operation select_next_controller() completes within finite time and returns the id of a potential controller other than the caller.*

2.2.3 Description of the Generalized Framework

The introduction of the two abstract types simplifies the presentation of framework and clearly separates the orthogonal concerns of controller migration and the scheduling of critical section entry requests. The algorithm at *node i* is described by the algorithm for P_i and M_i; $1 \leq i \leq n$.

Algorithm for client process P_i: A client process communicates with its associated MUTEX server by using messages **request_cs_entry**, **exit_cs**, and **request_granted** as shown below.

```
begin
  :
  send request_cs_entry to M_i;
  wait until request_granted received from M_i;
  execute critical section code
  send exit_cs to M_i;
  :
end
```

Algorithm for MUTEX server M_i: The algorithm for a MUTEX server is described in terms of its three roles - *requestor*, *auxiliary*, and *controller*. These roles can be concurrently active in a server and communicate among each other by message passing as well as by using shared variables.

The roles are described as a collection of event-action pairs. The actions in event-action pairs are executed atomically.

Each server maintains a global variable *current_controller*, which holds the identity of the current controller as known to the server. Initially, the variable *current_controller* at all nodes correctly refers to the *id* of the node whose controller role is active. All other nodes except the *current_controller* have their controller role deactivated. The objects REQUESTS_INFO and MIGRATION_INFO are implemented by additional global variables and by enhancing the roles by extra event-action pairs.

- Requestor Role: The request of a client is forwarded to M_i by invoking the *register_request* operation. Also, the *requestor* role interacts with the *controller* role, whether local or remote by receiving *request_granted* message and subsequently sending *exit_cs* message. The controller migrates only in between critical section entries. The algorithm for the requestor role is as follows:

```
/* Requestor role at M_i */

Upon receipt of request_cs_entry
      register_request(i);
```

Upon receipt of *request_granted*(j)
current_controller := j;
send **request_granted** to P_i;
Upon receipt of **exit_cs**
send *exit_cs* to *current_controller*;

- Controller Role: The *controller* role has primarily two responsibilities: scheduling critical section entry requests and eventually relinquishing the *controller* role to another MUTEX server. The algorithm for *controller* role is as follows:

/* Controller Role at M_i */

Upon receipt of *exit_cs*
if *test_migration_condition*() **then**
k := *select_next_controller*();
update_controller_info(k);
send *become_controller* to M_k;
current_controller := k;
/* deactivate controller role */
else
j := *select_request*();
if $j \neq$**null then** send *request_granted*(i) to M_j;
Upon receipt of *wake_up*
j := *select_request*();
if $j \neq$**null then** send *request_granted*(i) to M_j;

- Auxiliary Role: In Auxiliary role, a server participates in the implementation of REQUESTS_INFO object and in the migration of controller responsibility. The algorithm for auxiliary role at this stage is as follows:

/* Auxiliary role at M_i */

Upon receipt of *become_controller*
current_controller := i;
/* activate controller role */
send *wake_up* message to M_i(itself);

2.3 Correctness

A mutual exclusion algorithm is correct if it ensures that each request for critical section entry is granted within finite time and atmost one process is in the critical section at any instant. Stronger fairness requirements may be imposed for scheduling critical section requests. At the framework level, we are concerned with only starvation and deadlock freedom in scheduling entry requests. Assuming that properties 1-7 are satisfied, we show the correctness of the generalized framework in [8].

3 A Taxonomy of Migrating Controller Based Algorithms

From the framework described in the previous section, a spectrum of algorithms can be derived by various implementations of REQUESTS_INFO and MIGRATION_INFO objects.

3.1 Implementation of REQUESTS_INFO

The primary information encapsulated in REQUESTS_INFO is the collection of pending requests. Depending on where this collection is stored, *register_request*() may have to route the request accordingly. This requires routing information which too is part of REQUESTS_INFO. The operation *update_controller_info*() is invoked when migration of the controller takes place, for updating the routing information if it depends on the location of controller. Several alternatives for maintaining the pending requests collection are outlined next.

3.1.1 Centralized Queue

In the centralized queue approach, the pending requests are stored in a queue at the controller node. The routing information is simply the identity of the current controller. The *register_request*() operation involves sending the request to controller node where the request is enqueued. The operation *select_request*() dequeues a request from the front of the queue. The operation *update_controller_info*() informs all MUTEX servers about the new controller.

A server's knowledge of the current controller may be outdated for two reasons:

- Concurrent execution of *register_request* and *update_controller_info*.
- Selective update by *update_controller_info*.

The implementation of *register_request* relies on the implementation of *update_controller_info*. There are three broad choices for *update_controller_info* - inform none, inform all, and selective inform.

- The "inform none" refers to no action in *update_controller_info*. The operation *register_request* in this case is implemented by either broadcasting the request to all servers or by propagation among servers until the request reaches the controller.
- In the "inform all" case, *update_controller_info*() operation broadcasts the identity of the new controller to all servers. If a non-controller server (usually the last controller) receives a request, it forwards the request to the correct current controller.
- In "selective inform" strategy, the identity of the new controller is communicated to only a subset of servers. Thus, some servers may not have the correct knowledge of the current controller and their requests may require several message exchanges to reach the controller.

3.1.2 Distributed Queue

In the distributed approach, the collection of pending requests is implemented as a linked list of entries. Each entry in the list refers to the *id* of the server next in queue. The *select_request* operation refers to the head of the queue, whereas the operation *register_request* enqueues at the tail of the queue. An entry corresponding to a request is stored at the originating node itself. The tail information is stored at the controller node. Thus, *register_request* will have to communicate with the controller to get tail information and subsequently update it. For *update_controller_info*, we have the same choices available as in the centralized case. Alternative implementations of distributed queue are also possible.

3.1.3 Partially Ordered Collection

In both approaches above, the pending requests are totally ordered. There are alternatives where an explicit total order on requests is not imposed until the scheduling time. The partially ordered collection is not stored on a single node but partitioned into sub-collections which are stored as queues on distinct nodes. These sub-collections may be inter-related in an hierarchical fashion to form a tree structure. The operation *register_request* routes the request to the appropriate level in the hierarchy and the operation *select_request* picks up request from the hierarchy in a manner so as to avoid starvation. The hierarchy may be static or dynamic.

3.1.4 Ring Structure

The ring structured approach is a particular case of partially ordered collection approach. The ordered sub-collections in this case are arranged in the form of a ring rather than a tree. The operation *register_request* inserts a request in a sub-collection, which is a queue at a distinct node. The *select_request* operation schedules requests by traversing the ring.

3.1.5 Replicated Collection

Yet another approach is to have replicated copies of pending requests queue at multiple nodes. In this case, *register_request* must insert the request in all copies and *select_request* must remove the request from all copies. Further, if the nodes maintaining copies dynamically change (e.g. when the controller migrates) then *update_controller_info* must update the routing information to reflect the above change.

3.2 Implementation of MIGRATION_INFO

Controller migration is the delegation of controller role from one MUTEX server to another and involves three issues:

- When to migrate? The operation *test_migration_condition*() determines when to migrate.
- Where to migrate? The operation *select_next_controller*() selects the next controller.
- How to migrate? Implementation involves invoking *update_controller_info* and sending a *become_controller* message to new controller.

The first two issues are mutually independent of each other. We now outline implementation alternatives for the two operations on MIGRATION_INFO. An implementation of MIGRATION_INFO is derived by selecting an implementation for each of the two operations from the available alternatives.

3.2.1 Operation *test_migration_condition*()

This operation affects the frequency of migration. First, we consider two extreme cases: *No migration* and *Migration* with each exit from critical section.

- In the case of "No migration", *test_migration_condition* is implemented to always evaluate to **false**. Alternatively, the *controller* role can be modified to skip the test for migration altogether.
- If migration is as frequent as critical section entries then *test_migration_condition* is implemented to always return **true**. Again, the same effect can be achieved by skipping the test altogether.

The case where migration is less frequent than the critical section entries requires maintaining some state information which is updated with each invocation of *test_migration_condition*(). We provide two simple choices for controlling the frequency of migration:

- In *counter-based* approach, MIGRATION_INFO maintains a counter *req_served* which is incremented each time *test_migration_condition*() is invoked. When the value of *req_served* equals a threshold value **max_req**, *test_migration_condition*() returns **true**, otherwise it returns **false**. The new controller starts by initializing *req_served* to zero.
- In *timer-based* approach, a timer is started with a predetermined value **max_time** when a server assumes controller role. MIGRATION_INFO maintains a boolean variable *migrate* which is set to **false** when migration takes place. When the migration timer expires, *migrate* is set to **true**. A subsequent invocation of *test_migration_condition* then returns **true**.

3.3 Taxonomy

In the taxonomy shown in table 1, single controller based algorithms for mutual exclusion are laid out in a two dimensional space. The two dimensions pertain to the implementation of REQUESTS_INFO and MIGRATION_INFO objects respectively.

Each category in table 1 can be further sub-divided among sub-categories to highlight the differences between algorithms falling in the same category.

From table 1, it can be seen that most of the existing algorithms appear in only few categories, whereas there are no known algorithms in several other categories. In the next section, we present two new algorithms CDC and DDC which appear in categories as shown in table 1. It should also be noted that not all categories will lead to attractive algorithms, from performance point of view.

	No Migration	*Static Routing*		*Dynamic Routing*	
		Each CS Exit	Counter / Timer	Each CS Exit	Counter / Timer
Centralized Queue	[6]			[12]	CDC*
Distributed Queue		[4]		[1], [5] [10], [11]	DDC*
Hierarchy				[2], [3] [9], [7]	
Ring		[4]			
Replicated Queue					

Table 1. *Classification of Single Controller Based Distributed Mutual Exclusion Algorithms*

4 Derivation of Algorithms

We now present two new algorithms to demonstrate the derivation of algorithms from the framework.

4.1 CDC

(Centralized queue, Dynamic routing, Counter-based migration).
In this algorithm instead of migrating with each exit from critical section, the migration frequency is less and is controllable by setting the parameter **max_req**.

4.1.1 Implementation of REQUESTS_INFO

Data Structures:

At each MUTEX server
current_controller : *id* of current controller maintained in each MUTEX server;
At controller node only
request_queue : queue of requests, initialized to empty;
cs_status: (**free**, **busy**), initialized to **free**;

Operations:
All actions shown below are for the server M_i.

- *select_request*() /* invoked in controller role */
  {
  if *request_queue* $\neq \emptyset$ **then**
  {
  cs_status := **busy**;
  return first entry dequeued from *request_queue*;
  }
  else
  cs_status := **free**;
  return **null**;
  }

- *register_request* involves executing procedure *register_request()* in requestor role and subsequently performing an action in controller role and also possibly in the auxiliary role.

  In requestor role
  register_request(i) { send *request_cs_entry*(i) to *current_controller*;}
  In controller role
  Upon receipt of *request_cs_entry*(k)
  enqueue k in *request_queue*;
  if *cs_status* = **free** **then** send *wake_up* to M_i (itself);
  In auxiliary role
  Upon receipt of *request_cs_entry*(k)
  /*Controller role not active */
  send *request_cs_entry*(k) to *current_controller*;

- *update_controller_info* requires actions in controller role of controller and the auxiliary role of other servers.

  In controller role
  update_controller_info(k) {send *new_controller*(k) to all servers except k;}
  In auxiliary role
  Upon receipt of *new_controller*(k)
  current_controller := k;

4.1.2 Implementation of MIGRATION_INFO

Data Structures:

Global parameters - at each MUTEX server
candidate_controllers: list of servers which can assume controller role;
max_req: max number of consecutive requests to be served by the controller.
At controller node only
req_served : number of consecutive requests served by the controller, initialized to zero.

Operations:

- *test_migration_condition*()
  {
  req_served := *req_served* + 1;
  if (*req_served* = **max_req**) **then**
  req_served := 0;
  return **true**;
  else return **false**;
  }

- *select_next_controller*()
  {
  j := first entry in *request_queue*;
  k := second entry in *request_queue*;
  if k = **null** or $k \notin$ *candidate_controllers* **then**
  k := any node other than i, j in *candidate_controllers*;
  return k;
  }

4.1.3 Correctness of CDC

The correctness of CDC is shown by proving that the implementation of REQUESTS_INFO and MIGRATION_INFO satisfy the properties 1-7. The properties 2, 3, 4, 6 and 7 directly follow from the pseudocode description. The operation *register_request* involves sending the message *request_cs_entry* to the *current_controller* which receives the above message within finite time and consequently enqueues the corresponding request. In the event where controller is being migrated while *register_request* message is in transit, this message will be routed by old controller to the new controller. Thus, property 1 is ensured. Further, property 5 is ensured since the

only effect of concurrent execution of *update_controller_info* and *register_request* is the delay in receipt of *request_cs_entry* by the controller because of re-routing.

4.1.4 Complexity

The message overhead per critical section entry is 3, not counting 3 local messages from client to server and assuming that controller does not migrate while the request is in transit. Extra message will be required for re-routing the *request_cs_entry* message in case if the controller migrates in between. Further, $n-1$ messages are required per migration. The average message overhead per critical section entry (including migration) is $c + (n-1)/\mathbf{max\_req}$, where c is the sum of 3 and the rerouting overhead. Thus, average message overhead per critical section entry is reduced as **max_req** is increased. If **max_req** is chosen to be 1, the controller migrates with each critical section entry and the message overhead is $(n-1)+c$. On the other hand, if $\mathbf{max\_req} = \infty$, the algorithm reduces to the non-migrating centralized controller and the message overhead is 3 per critical section entry.

4.2 DDC

(Distributed queue, Dynamic routing, Counter-based migration)
This algorithm differs from CDC algorithm in just one aspect. Here, instead of storing the requests at the controller node, we store them in a distributed manner among the requesting nodes. Each requesting node has a variable *next_in_q* (initialized to **null** by *register_request*), which, if not **null**, refers to the node whose request immediately follows the former node and is scheduled in that order. The controller node keeps track of the node whose request is last, in a variable *last_in_q*. Also, the controller maintains a variable *first_in_q*, which, if defined, refers to the node whose request is first in queue. Since the controller does not store the complete queue, it must get the identity of the server to be granted next from the exiting server. This is implemented by piggybacking *next_in_q* to *exit_cs* message. The *last_in_q* server is sent a message *next_requestor* by the controller to allow the server to update it's value of *next_in_q* when another request arrives at the controller. Migration is similar to CDC algorithm and thus, the migration frequency is controllable by parameter **max_req**.

4.2.1 Implementation of REQUESTS_INFO

Data Structures:

At each MUTEX server
current_controller: controller *id* as known
next_in_q: node identifier - if not **null**, refers to node whose request immediately follows this server's request;
At controller node only
last_in_q: node identifier - refers to the node whose request is last among the pending requests; intialized to **null**;
first_in_q: node identifier - if not **null**, refers to the node whose request was granted last; initialized to **undefined** (null);
cs_status: (**free**, **busy**), intialized to **free**;

Operations:

- *select_request*(*next_in_q*)
  /* invoked in controller role - *next_in_q* is received with *exit_cs* message*/
  {
  if *next_in_q* $\neq$ **null then**
  if *next_in_q* = *last_in_q* **then** *last_in_q* := **null**;
  cs_status := **busy**;
  return *next_in_q*;
  else /**next_in_q* = **null** */
  if *first_in_q* = **undefined then**
  cs_status := **free**;
  return **null**;
  else
  cs_status := **busy**;
  next_in_q := *first_in_q*;
  first_in_q := **undefined**;
  return *next_in_q*;
  }

- *register_request* involves performing actions in requestor role, controller role and also possibly in the auxiliary role.

  In requestor role
  register_request(*i*)
  {
  next_in_q := **null**;
  send *request_cs_entry*(*i*) to *current_controller*;
  }
  Upon receipt of *next_requestor*(next)
  next_in_q := next;
  In controller role
  Upon receipt of *request_cs_entry*(*k*)
  if *last_in_q* $\neq$ **null then**
  send *next_requestor*(*k*) to *last_in_q*;
  last_in_q := k;
  else
  last_in_q := *k*;
  if *cs_status* = **free then** send *wake_up* message to M_i (itself);
  In auxiliary role
  /* executed only when controller role is not active */
  Upon receipt of *request_cs_entry*(*k*)
  send *request_cs_entry*(*k*) to *current_controller*;

- The implementation of operation *update_controller_info* is same as in CDC algorithm.

4.2.2 Implementation of MIGRATION_INFO

The implementation of MIGRATION_INFO is identical to the CDC algorithm except for a minor change in the operation *select_next_controller*.

- *select_next_controller*()
  k := *last_in_q*;
  if (k = **null** or $k \notin$ *candidate_controllers*) **then**
  k := any node other than i in *candidate_controllers*;
  return k;
  }

4.2.3 Correctness of DDC

From the description of *register_request*, it is clear that its invocation by M_i results in setting *last_in_q* to i, at the controller in finite time and if *cs_status* is **free**, *wake_up* message is sent to the controller. Thus properties 1 and 2 are ensured. In *select_request* the truth of conditions, *next_in_q* = **null** and *first_in_q* = **undefined** implies that there are no pending requests, in which case *cs_status* is set to **free** and **null** is returned. In other cases, *cs_status* is set to **busy** and the request at the top of the queue is returned. The requests are served in the order in which they were received at the controller. Thus property 3 is ensured. Other operations are implemented as in CDC algorithm and their proofs are obvious from their description.

4.2.4 Complexity

The message overhead per critical section entry is 3 to 5 depending on whether controller migrates during submission of request or whether the next request is submitted before or after the server gets a grant. Further, as in CDC, $n-1$ messages are required per migration. The total average message overhead per critical section entry (including migration) is $c+(n-1)/\mathbf{max\_req}$, where c is 3 or 5. Thus, message complexity of algorithms DDC and CDC are identical. However the algorithm DDC is more amenable to an extension to a fault-tolerant version [1] than the CDC algorithm because of a distributed queue implementation.

5 Conclusion

A new framework for migrating controller based distributed mutual exclusion algorithms is presented, which separates the orthogonal concerns of migrating the controller and the granting of critical section entry. An interesting aspect of the framework is its layered approach using distributed abstract types to encapsulate migration and request scheduling concerns and provides a basis for taxonomical classification of single controller based distributed mutual exclusion algorithms.

References

[1] R. K. Arora, S. P. Rana, and N. K. Jain. Achieving mutual exclusion in a distributed computing environment. *Information Systems*, 7(4):359–365, 82.

[2] J.-M. Helary, A. Mostefaoui, and M. Raynal. A general scheme for token and tree based distributed mutual exclusion algorithms. Technical report, IRISA Campus de Beaulicu, France, 92.

[3] J.-M. Helary, N. Plouzeau, and M. Raynal. A distributed algorithm for mutual exclusion in an arbitrary network. *The Computer Journal*, 31(4):289–295, 88.

[4] A. J. Martin. Distributed mutual exclusion in a ring of processes. *Science of Computer Programming*, 5:265–276, 85.

[5] M. Mizuno, M. L. Neilsen, and R. Rao. A token based distributed mutual exclusion algorithm based on quorum agreements. In *Proceedings of the 11th International Conference of Distributed Computing Systems*, pages 284–299, 91.

[6] C. Mohan and A. Silberschatz. Distributed control - is it always desirable? In *Proceedings of the Symposium on Reliability in Distributed Software and Database Systems*, pages 102–106, 81.

[7] M. L. Neilsen and M. Mizuno. A dag based algorithm for distributed mutual exclusion. In *Proceedings of the 11th International Conference of Distributed Computing Systems*, pages 284–299, 91.

[8] S. Rana, K. Raman, and V. Chaudhary. Migrating controller based algorithms for mutual exclusion in distributed systems. Technical report, Parallel and Distributed Systems Laboratory, Department of Computer Science, Wayne State University, Detroit, MI 48202, 1994.

[9] K. Raymond. A tree-based algorithm distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 89.

[10] G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 81.

[11] M. Singhal. On the application of ai in decentralized control: An illustration by mutual exclusion. In *In Proceedings of 7th International Conference on Distributed Computing Systems*, pages 232–239, 87.

[12] I. Suzuki and T. Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, 85.