Region-based Software Distributed Shared Memory

Song Li, Yu Lin, and Michael Walker

CS 656—Operating Systems

May 5, 2000

Abstract

In this paper, we describe the implementation of a software-based DSM model that supports variable-sized shard "regions", where a region is a logical page that can be adjusted according to the access patterns of the user program. Our project supports the multiple reader, single writer (MRWS) replication algorithm, and uses a least-recently-used (LRU) region replacement model. We found that our DSM model can yield better performance that an equivalent RPC model of communication, and can also be easier to use.

1 Introduction

Despite the advances in processor design, users still demand more performance. Eventually, single CPU technologies must give way to multiple processor parallel computers: it is less expensive to run 10 inexpensive processors cooperatively than it is to buy a new computer 10 times as fast. This change is inevitable, and has been realized to some extent in the specialization of subsystems like bus mastering drive controllers. However, the need for additional computational power has thus far rested solely on advances in CPU technologies.

In distributed systems, there are two kinds of fundamental interprocess communication models: shared memory and message passing. From a programmer's perspective, shared memory computers, while easy to program, are difficult to build and aren't scalable to beyond a few processors. Message passing computers, while easy to build and scale, are difficult to program. In some sense, shared memory model and message passing model are equivalent.

One of the solutions to parallel system communication is Distributed Shared Memory (DSM), where memory is physically distributed but logically shared. DSM appears as shared memory to the applications programmer, but relies on message passing between independent CPUs to access the global virtual address space. Both hardware and software implementations have been proposed in the literature. The advantages of DSM programming model are well known. Firstly, shared memory programs are usually shorter and easier to understand than equivalent message passing programs. The memory accessing-paradigm is already popular with users. Secondly, shared memory gives transparent process-to-process communication, and it can remove the user from any explicit awareness of communication if desired. Also, programming with shared memory is a well-understood problem.

Our team has developed a version of the DSM programming model as our distributed operating systems course project. The objective of this project is to develop an understanding of the theory and implementation of modern DSM. This has included a detailed study of caching and cache consistency models, page replacement strategies, etc. We provide a simple interface for programmers that wish to use the DSM model for communication and sharing between processes in a loosely-coupled system.

We describe the design of our distributed shared memory model in section 2. We explain the reasoning behind our design decisions, and compare them to other possible solutions

used in previously done work in DSM. Section 3 describes our implementation in some detail, including the basic mode of operation. Section 4 provides an evaluation of our work, comparing it to RPC communication in terms of performance and ease-of-use. We conclude with a discussion of future work (section 5) that might be implemented to improve the performance and robustness of our DSM system. Section 6 concludes with a summary of our work.

2 DSM Design

Our design decision began with a careful review of typical DSM models, as well as the current research in the area. We chose to implement a software-based DSM model that supports variable-sized shared "regions", where a region is a logical page that can be 4adjusted according to the access patterns of the user program. Our project supports the multiple reader, single writer (MRSW) replication algorithm, and uses a least-recently-used (LRU) region replacement model. A detailed description of the design follows.

Our DSM model shares "regions" of memory between nodes in a loosely-coupled system. We based this idea on a similar concept used by Johnson, Kaashoek, and Wallach at MIT [JKW95]. We chose this approach over a shared variable or shared object approach both for simplicity and usefulness.

A region is a variable-size piece of memory that is similar to a segment. The general idea behind region-based shared memory is similar to page-based shared memory, except that region size can change, whereas page size is usually determined statically, and often by the underlying system architecture. We chose to use variable-sized regions rather than a fixed-size page because there are advantages to every granularity of a region, and a flexible system should provide different levels of such granularity. For instance, using large-sized regions greatly reduces the amount of overhead in sending data between nodes because requests for regions are kept to a minimum, and with caching implemented, very few requests must be made to the shared memory provider. However, a small-sized region has the advantage of lessening the chance that two shared variables will lie on the same region, which makes cache invalidation less frequent.

Different user programs have different properties, and a region-based system allows the user to specify the desired level of granularity. However, we provide a region size by default that has been proven to be effective for most applications. Our philosophy about regions is similar to the Legion research project philosophy. Instead of choosing one policy, we allow the user to adjust the policy to their liking, and we provide reasonable defaults.

In order to make our shared memory useful to multiple nodes in a distributed system, we chose to allow multiple readers access to the same region of shared memory. This improves performance and saves space by allowing multiple processes on possibly different nodes to work together in the same logical shared address space.

However, we chose to implement the multiple readers, single writer (MRSW) replication policy in order to provide a consistency model that allows for some caching without an enormous amount of complication. We based this idea on an algorithm provided by Stumm and Zhou [SZ90]. Details about our implementation of the MRSW model can be found in section 3.

We also needed to choose a policy for region replacement when there is not enough shared memory to hold all regions resident in memory at once. We chose a simple and effective LRU replacement strategy. Information about the last usage of a region is held and used to determine which region has not been accessed lately. The least used region is chosen as the candidate for replacement. If there is room on the memory provider's disk, we store the region to disk. If not, we migrate the region to another provider, in hopes that it will have space. Finally, if all providers have run out of memory and disk space, then our system is exhausted. The region will continue to be passed around, waiting for a space, until another region is freed. Because this method provides a reasonable replacement strategy that ensures that the system is used to its limit, and because it is relatively simple to implement, we chose the LRU model for region replacement.

2.1 General behavior

We divide our DSM model into three logically separate entities: client, provider, and manager. A discussion of the basic interaction between these three entities follows.

A node that requests allocation of shared memory is called a **client**. A client can request multiple regions for allocation without having to spend any computational time choosing where the allocation will take place. The client simply submits all requests to a manager. The **manager**'s job is to handle requests from clients, allocating memory appropriately and giving the client a way to reference such memory. The available shared memory is determined by the amount donated by providers. A **provider** contacts a manager to volunteer memory, and then allows clients access to that memory.

It is important to note that there is no reason a node cannot be both a client and a provider, or even a representation of all three logical entities. However, we made this logical distinction to draw a clean line between the three jobs in our model.

We defined the different requests allowed on shared memory regions by creating the client interface model:

```
r_handle sm_malloc(size);
int sm_regionat(r_handle, attrib);
int sm_read(r_handle, offset, buf, size);
int sm_write(r_handle, offset, buf, size);
int sm_regiondt(r_handle);
int sm_free(r_handle);
```

The interface is modeled after UNIX shared memory system calls, and generally operate under the same principles, but in a distributed system. A region is first created outside the address space of any process using sm_malloc(), and a region handle r_handle is returned. This handle can be used by any client to logically attach the memory to their address space by using sm_regionat(). After the region is attached, the client may perform read and write operations on it, just like it was using system calls to read and write locally available space, by using sm_read() and sm_write(). The details of the read and write operations are abstracted by the interface, and, as far as the client is concerned, the memory accesses appear to be call-by-reference. Of course, the underlying operations implement a call-by-value message-passing system between client and provider, but the client does not have to be explicitly aware of this communcation.

When a client is finished with the allocated regions, it is responsible for detaching the region from its address space by calling sm_regiondt(), and optionally freeing the space (sm_free()). The client may choose not to free the space if it no longer requires the region, but another client is still using it

3 DSM Implementation

In our DSM implementation, any communication messages between any two entities must be passed in a GLOBAL_MSG data structure, and any data-passing of regions must be passed through a DATA_MSG. By allowing only one message of each type, we were able to standardize and simplify communication.

```
struct GLOBAL_MSG
{
  int label;
  int type;
  int size;
  int index;
  int result;
  int attr;
  int rg_handler;
  int item_count;
  struct region_item rg_item[MAX_PG_ITEM_COUNT];
};
```

In a global message, the label always identifies the entity type of the sender and receiver. A message type identifies the purpose of the message, and which fields in the remainder of the message are relevant. For example, a message with label "C2M" and type "MALLOC" would indicate a memory allocation request from a client to a manager. The following fields are optionally used to specify the details of the communcation message.

Instead of using the same message structure to pass regions, all region data is passed in a separate data message.

```
struct DATA_MSG
{
  int label;
  int type;
  int index;
  int result;
  int attr;
  int rg_handle;
  char data[REGION_SIZE];
};
```

The data message holds the same identifying fields as the global communication message, and also includes a region handle and a data array. A region handle is the unique index into the entire list of regions maintained by the manager. The data array is exactly one region large. Note that one and only one region can be passed at a time using the data message. This enforces the size restriction policy as defined by REGION_SIZE, ensuring that the granularity of all data passing is similar.

3.1 Basic Operation

Now that the basic entities in the model, the interface for memory allocation, and the underlying message structure has been defined, we can describe the basic mode of operation.

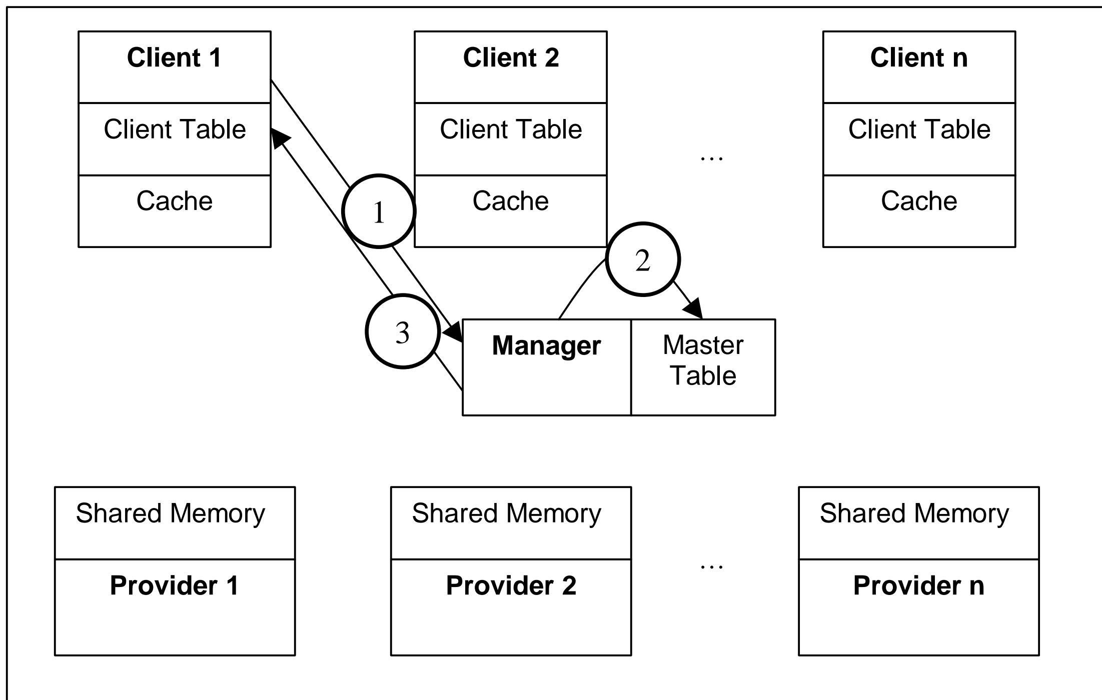

Figure 1a. The client initiates a request to allocate a number of shared memory regions, and receives a region handle from the manager.

In figure 1a, the client passes the request to allocate memory via the sm_malloc() and sm_regionat() calls to the manager (step 1). The manager then queries its table of available memory providers, and makes an allocation decision. The manager then creates a region handle for the new memory region, and enters the handle into the table in step 2. Although the details of the handle are not important, it is worth noting that the handle serves as a unique identifier for the location of a specific region in the distributed shared memory model. After the handle has been recorded, the manager passes it, along with a success message, to the client so that it can reference the newly allocated memory. The handle is attached to the client's memory by being placed in the client region table.

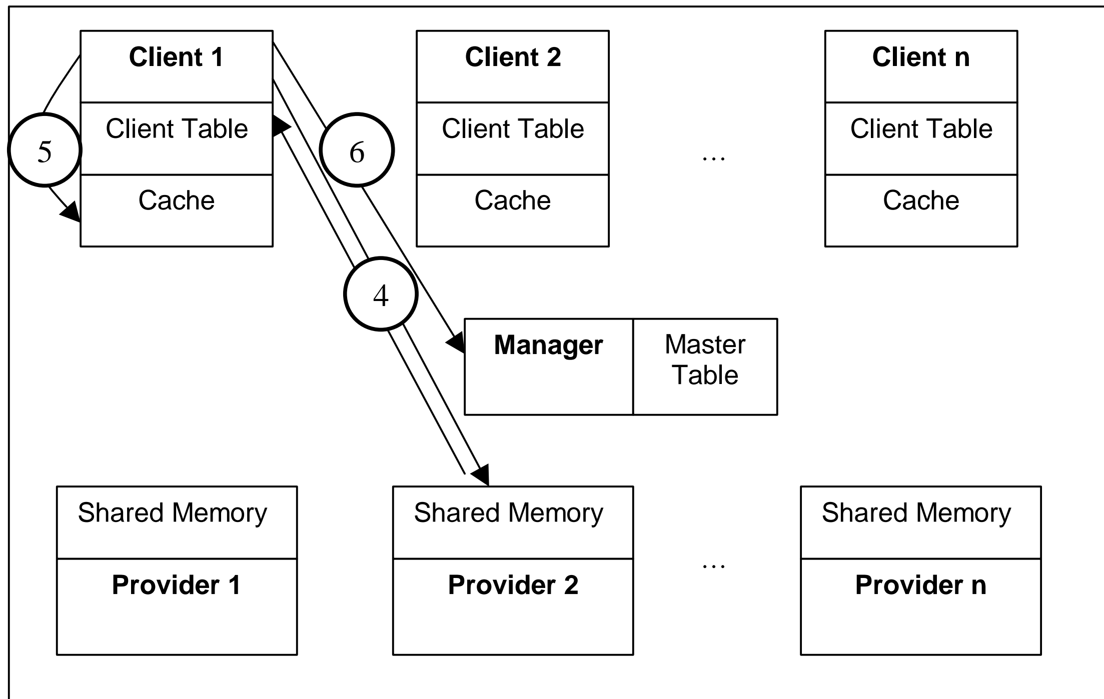

Figure 1b. The client accesses, and optionally caches, shared memory via the shared memory provider, and reports to the manager when it is finished.

In figure 1b, the client uses the region handle to access shared memory from a provider (step 4). Note that all shared memory access communcation is between the client and provider, and the manager is not involved. Because the manager already has sufficient information for keeping track of region accesses, it can be removed from the accessing operations, reducing the communcation overhead. The client can then perform sm_read() and sm_write() operations on the region, which is not shared in this simple example. For an explanation of consistency issues when the region is shared, see section 3.2.

In this simple model, the client has the option of caching a copy of the region, and performing operations on the region locally (step 5). This is advantageous because it reduces network communication costs greatly, and allows faster memory accesses.

When the client is finished with the shared memory regions it allocated, it notifies the manager (step 6) using the sm_detach() call. The manager removes the client from the list of users of that region, but may leave the region open if (1) the client has not called sm_free(), and (2) there are other users of the region in the master table.

3.2 Replication and Cache Consistency

In order to improve memory access speed, our model allows region caching under certain conditions. Region replication is beneficial because access to a cached local copy of a region requires no network communication, which is often the source of the greatest overhead in distributed memory accessing. Therefore, we allow multiple readers to cache copies of a region as long as their access patterns are read-only. When another client wants write access to such a region, all cached copies must be invalidated. We restrict the number of concurrent writers to one, because consistency between multiple readers and multiple writers is an overly complicated problem for the type of service our system provides.

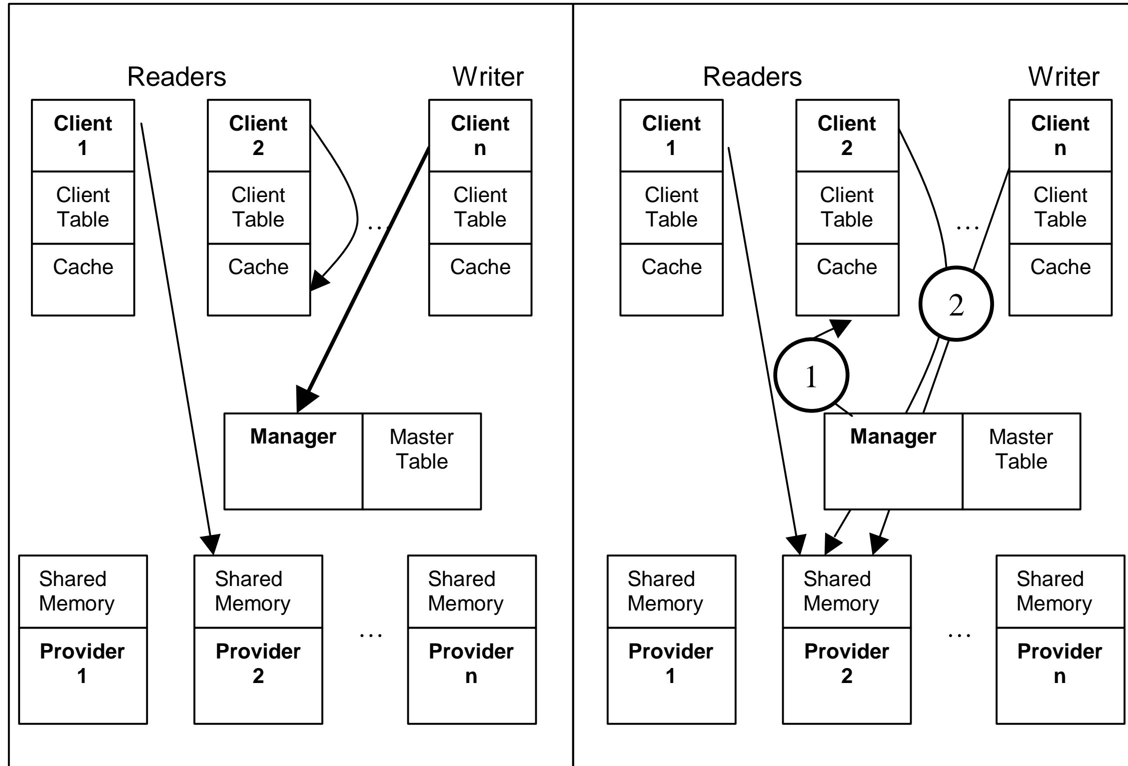

Figure 2. The MRSW replication model. In 2a, a write request has been issued for a region that is currently being read by two or more clients. 2b illustrates the changes the manager initiates to support consistency in the MRSW region access pattern.

Figure 2a depicts a typical scenario when a write request is processed in the presence of multiple readers. One client (in this case, client 1) may be directly accessing the shared memory, whereas another client (in this case, client 2) may be working with a cached copy. Because reads typically dominate region access, this scenario is fairly common. Upon a request for write access to the shared region, the manager must invalidate any cached copies of the region by sending a message to all clients working with replications (figure 2b, 1). After the client caches have been invalidated, the writer is allowed to access the shared region. All read/write accesses must be made via the shared memory provider (2).

3.3 Region Replacement Strategy

We developed a region replacement strategy for occasions when the manager does not have enough provided shared memory space to allocate another region, and a region must be evicted. The strategy is two-fold: first, replace regions that are cached and only read by clients; second, replace the least recently used region.

When a region must be evicted from a provider, the manager hopes to find a region that is currently attached as read-only access, and is cached. In this case, the manager sends a message to all clients working with non-cached copies of the region, known as an "update" message. These clients now direct all memory accesses to the client with the cached copy, rather that the provider directly. Meanwhile, the manager replaces the original; region with the newly allocated space. Thus, both the old region users and the new region request have been satisfied.

If the first situation is not applicable, the manager evicts the least recently used region, and requests that provider of that region to write the region to disk. This is expensive, and is only done as a last resort. However, evicting the least recently used region is usually a simple and effective method for minimizing the total cost of eviction for the overall system. If would not be effective when the current number of read/write regions in use are larger than the number of available shared memory regions and the access patterns are random. However, random access on shared memory is not the normal behavior, and nearly all replacement strategies suffer in this particular scenario.

4 Evaluation

In our evaluation, we tested the performance of our region-based DSM model against an RPC-based model under similar conditions. Generally, we found that our model had a large startup overhead, but while running, had a significantly smaller communication cost than the equivalent RPC-based solution.

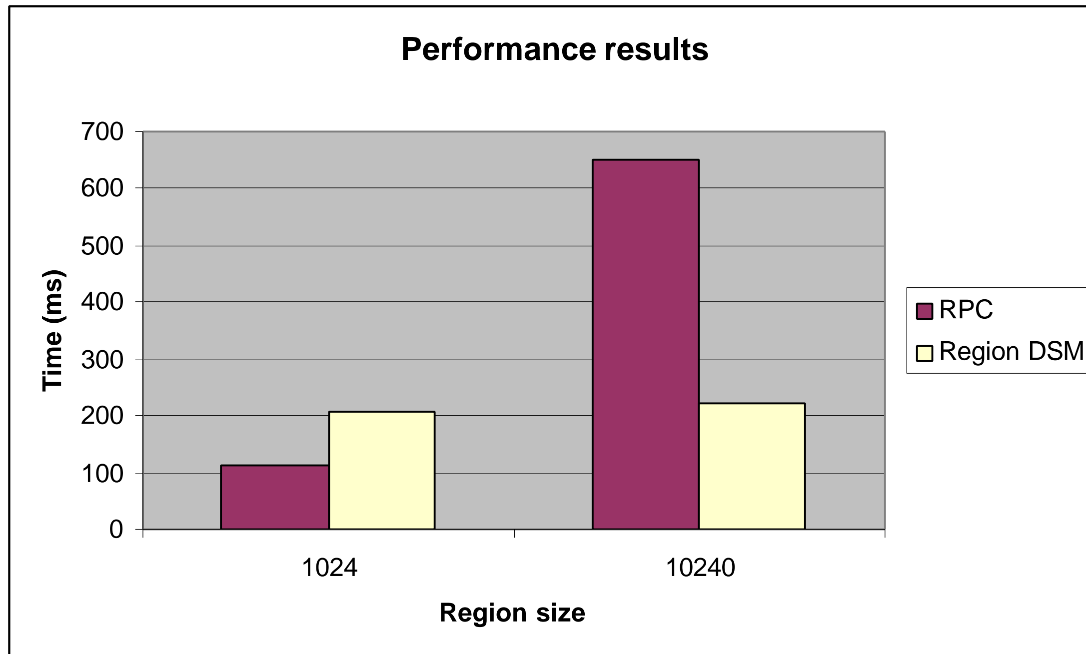

Figure 3. Testing region-DSM and RPC cost for read/writes to different-sized regions.

We found that for our test application, the region-based DSM model performed better than the RPC model when the region size was between 1K and 10K, and performed worse than the RPC model when the region size was <1K. This tested only the simplest DSM model we provided, and no caching was activated. Furthermore, this basic test does not test performance under multiple-reader situations. We believe that both of these cases will only improve our performance in most circumstances.

5 Future Work

Although our current DSM implementation is working and complete, certain additions to the system could potentially improve the system performance and robustness. Firstly, the idea of a region protection mechanism, similar to segment protection, could be used to protect from invalid memory access from misguided or malicious users. To protect against simple misuse, the provider would hold a list of hosts with valid access to each shared region it provides. Before sending data, the provider would check to see if the client has access to the region. In this way, accidental out-of-bounds accessing could be prevented.

However, this still would not protect against a malicious user that forcefully attempts access to regions. To prevent intentional out-of-bounds region access, the provider would hold a list of public keys corresponding to each valid client. Each client would hold a public key identifier, and would authenticate with the provider before accessing shared memory regions. This solution would greatly strengthen security, but potentially reduce performance, as the amount of computation for generating and verifying public keys can

be large. Ideally, the extra region protection could be activated at the user's request, allowing for variable levels of security based on the demands of the user.

Additionally, our system could benefit from manager replication. By replicating managers, we could potentially decentralize region management, which reduces the amount of communication any one manager must handle. This would also make our system more fault tolerant, because replicated manager data could persist in the case that one or more manager node fails. Because of the complex issues involved with this strategy, we have omitted it from our current implementation.

6 Conclusion

In this paper, we decsribe the implementation of a software-based DSM model that supports variable-sized shared "regions", where a region is a logical page that can be adjusted according to the access patterns of the user program. Our project supports the multiple reader, single writer (MRSW) replication algorithm, and uses a least-recently-used (LRU) region replacement model that takes advantage of region caching.

We found that our DSM model is especially valuable because it allows the user to define region size based on the needs of the application that uses DSM. The interface for users is similar to the UNIX shared memory model, and is easy to use. Furthermore, our region-based DSM model can yield better performance that an equivalent RPC model if communication. We believe that our model is useful in ease of use as well as performance when desiring interprocess communication in a loosely-coupled system.

References

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. London: McGraw-Hill, 1998.

[GS97] Kourosh Gharachorloo and Daniel Scales. Towards Transparent and Efficient Software Distributed Shared Memory. From *Proceedings of the Sixteenth Symposium on Operating Systems Principles*, pp. 157-169, 1997.

[JKW95] Kirk Johnson, Frans Kaashoek, and Deborah Wallach. CRL: High-Performance All-Software Distributed Shared Memory. From *Proceedings of the Fifteenth Symposium on Operating Systems Principles*, 1995.

[NL91] Bill Nitzberg and Virginia Lo. *Distributed Shared Memory: A Survey of Issues and Algorithms*. From IEEE Computer, pp. 52-60, 1991.

[S97] Pradeep Sinha. *Distributed Operating Systems: Concepts and Design*. New York: IEEE Press, 1997.

[SZ90] Michael Stumm and Songnian Zhou. *Algorithms Implementing Distributed Shared Memory*. From IEEE Computer, pp. 54-64, 1990.