Dashalyan Naidoo (v3c8) | Edward Huang (u5m8) | Kamil Wasty (n9a9) | Vincent Tan (g0l0b)

HiveMind: A Golang DSM Based on TreadMarks

09th March 2018

INTRODUCTION AND BACKGROUND

A distributed shared memory (DSM) system allows processes to assume a globally shared virtual memory even though they execute on nodes that do not physically share memory. Figure 1 illustrates a DSM system consisting of N networked workstation, each with its own memory connected by a network. The DSM software provides the abstraction of a globally shared memory, in which each processor can access any data item, without the programmer having to worry about where the data is or how to obtain its value. In contrast, in the “native” programming model on networks of workstations, *message passing*, the programmer must decide *when* a processor needs to communicate, with *whom* to communicate, and *what* data to send. For programs with complex data structures and sophisticated parallelization strategies, this can become a daunting task. On a DSM system, the programmer can focus on algorithmic development rather than on managing partitioned data sets and communicating values. In addition to ease of programming, DSM provides the same programming environment as that on (hardware) shared memory multiprocessors, allowing programs written for a DSM to be ported easily to a shared-memory multiprocessor. Porting a program from a hardware shared-memory multiprocessor to a DSM system may require more modifications to the program because the much higher latencies in a DSM system put an even greater value on locality of memory access.[3]

The programming interfaces to DSM systems may differ in a variety of respects. We focus here on memory *structure* and memory *consistency* model. An unstructured memory appears as a linear array of bytes, whereas in a structured memory processes access memory in terms of objects or tuples. The memory model refers to how updates to shared memory are reflected to the processes in the system. The most intuitive model of shared memory is that a read should always return the ”last value written”. Unfortunately, the notion of the ”last value written”, is not well defined in a distributed system. A more precise notion is sequential consistency, whereby the memory appears to all processes as if they were executing on a single multiprogrammed processor. With *sequential consistency*, the notion of the last value written, is precisely defined.

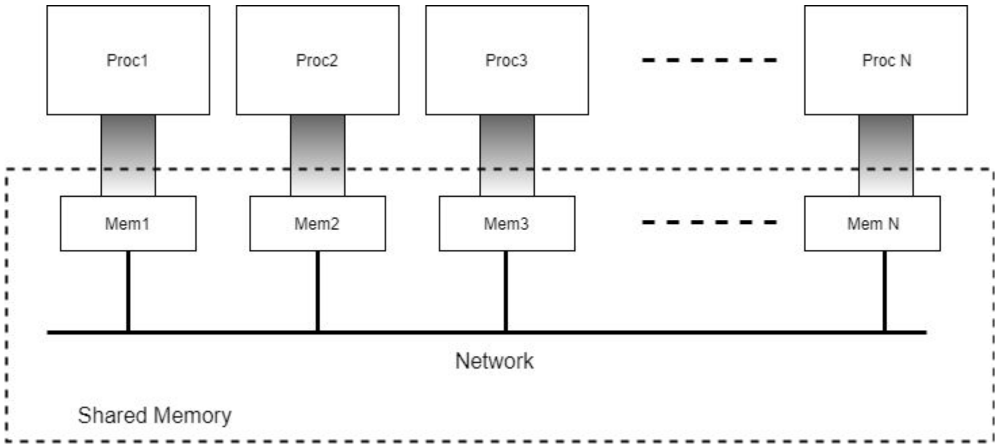

Figure 1: Distributed Shared Memory System from P. Keleher . *Distributed Shared Memory Using Lazy Release Consistency*, 1994.

The simplicity of this model may, however, exact a high price in terms of performance, and therefore much research has been done into *relaxed memory models*. A relaxed memory model does not necessarily always return to a read the last value written.

The system discussed in this paper, TreadMarks, provides shared memory as a linear array of bytes. The TreadMarks design focuses on reducing the amount of communication necessary to maintain memory consistency. To this end, TreadMarks presents a release consistent memory model to the user. The lazy implementation of release consistency in TreadMarks further reduces the number of messages and amount of data compared to earlier eager implementations. False sharing is another source of frequent communications in DSM systems. TreadMarks uses multiple-writer protocols to address this problem. Multiple writer protocols require the creation of diffs, data structures that record updates to parts of a page. With lazy release consistency, diff creation can often be postponed or avoided, a technique referred to as lazy diff creation by Keleher et al [4].

HiveMind is an implementation of the TreadMarks DSM as Golang package. We plan to implement the core features of TreadMarks, namely full API, LRC, and multiple-write with lazy diff creation. HiveMind runs at user level on Linux workstations. No kernel modifications or special privileges are required. As a result, this system is fairly portable. We plan to test HiveMind on a cluster of up to 8 nodes on Azure and run a Quicksort application as a benchmark.

The rest of this paper provides an overview of the TreadMarks architecture, the approach that we plan to take to implement and test our implementation, our development plan and SWOT analysis.

TreadMarks ARCHITECTURE

Each process in the TreadMarks DSM system allocates a portion of memory at the same position to be used as shared memory and the memory is divided into pages. A manager is assigned to each page and tracks the usage of their page. The core of TreadMarks is the utilization of a lazy release consistency (LRC) model for memory, where page writes only result in synchronization when another process is attempting to access it. At this time, the acquiring processor determines which modifications it needs to see according to the definition of RC.

Lazy Release Consistency

To do so, LRC divides the execution of each process into intervals, each denoted by an interval index. Every time a process executes a release or an acquire, a new interval begins and the interval index is incremented. Intervals of different processes are partially ordered: (i) intervals on a single processor are totally ordered by program order, and (ii) an interval on processor *p* precedes an interval on processor *q* if the interval of *q* begins with the acquire corresponding to the release that concluded the interval of *p*. This partial order can be represented concisely by assigning a vector timestamp to each interval. A vector timestamp contains an entry for each processor. The entry for processor *p* in the vector timestamp of interval *i* of processor *p* is equal to *i*. The entry for processor $q \neq p$ denotes the most recent interval of processor *q* that precedes the current interval of processor *p* according to the partial order. A processor computes a new vector timestamp at an acquire according to the pair-wise maximum of its previous vector timestamp and the releaser's vector timestamp.

RC requires that before a processor *p* may continue past an acquire, the updates of all intervals with a smaller vector timestamp than *p*'s current vector timestamp must be visible at *p*. Therefore, at an acquire, *p* sends its current vector timestamp to the previous releaser, *q*. Processor *q* then piggybacks on the release-acquire message to *p*, write notices for all intervals named in *q*'s current vector timestamp but not in the vector timestamp it received from *p*.

A write notice is an indication that a page has been modified in a particular interval, but it does not contain the actual modifications. The timing of the actual data movement depends on whether an invalidate, an update, or a hybrid protocol is used. TreadMarks currently uses an invalidate protocol: the arrival of a write notice for a page causes the processor to invalidate its copy of that page. A subsequent access to that page causes an access miss, at which time the modifications are propagated to the local copy.

An example to illustrate LRC is the following: consider variables x and y that are on the same page and processors P1, P2, and P3. Each processor executes the following operations:

```
P1: a1 x = 2 r1
P2:            a2 y = 3 r2
P3:                        a1 print x r1
```

What LRC does is that P3 only asks the previous holder of lock 1 for write diffs. If the operation were print x, y, P3 would print a stale value for y even though both variables are on the same page. The advantage is that less information flows through the network since information about y would be useless for P3 but the programmer must be careful to acquire the proper locks to avoid reading stale data. TreadMarks makes use of vector timestamps to overcome this problem, by requiring that processors need to send their current vector timestamp to previous lock holder and then having the previous holder sending write notices for all intervals (including updates to variables/pages that the new lock holder doesn’t care).

Multiple-Writer Protocols

False sharing occurs when two or more processors access different variables within a page, with at least one of the accesses being a write. A write to any variable of a page causes the entire page to become invalid at all other processors that cache the page. This leads to an access miss when any of these processors try to access the page and then the modified page is transported over the network. You can see that this is a problem since the original page would have sufficed, because the access was to a different variable than the one that was written.

Lazy Diff Creation

To address the problem of false sharing TreadMarks creates diffs lazily. A shared page is initially write-protected. At the first write, a protection violation occurs. The DSM software makes a copy of the page (a twin), and removes the write protection so that further writes to the page can occur without any DSM intervention. The twin and current copy can later be compared to create a diff. TreadMarks creates the diffs only when a processor requests modifications to a page or a write notice from another processor arrives for that page.

TreadMarks IMPLEMENTATION

Data Structures

Figure 2 details the major structures in Treadmarks: the page array, the proc array, the diff pool, write notice records and interval records. Each process will maintain its own version of these structures.

- The page array is the globally shared memory all the processes in the DSM system will be utilizing. There is one entry for each shared page. Each entry in the page array contains the following fields:
    - Current state: no-access, read-only, or read-write access
    - An approximate copyset specifying a set of processors that are believed to currently cache this page
    - For each page, an array indexed by processor of head and tail pointers to a linked list of write notice records corresponding to write notices received from that processor for this page.
- The proc array maintains a list of interval records the owning process knows about for each process in the system. An interval record is a record of a processes interaction with a page during a given length of time. Each of these interval records contains a pointer to a list of write notice records for that interval, and each write notice record contains a pointer to its interval record.
- The diff pool is a collection of diffs currently in the system, which is used to update pages that do not have the latest changes.

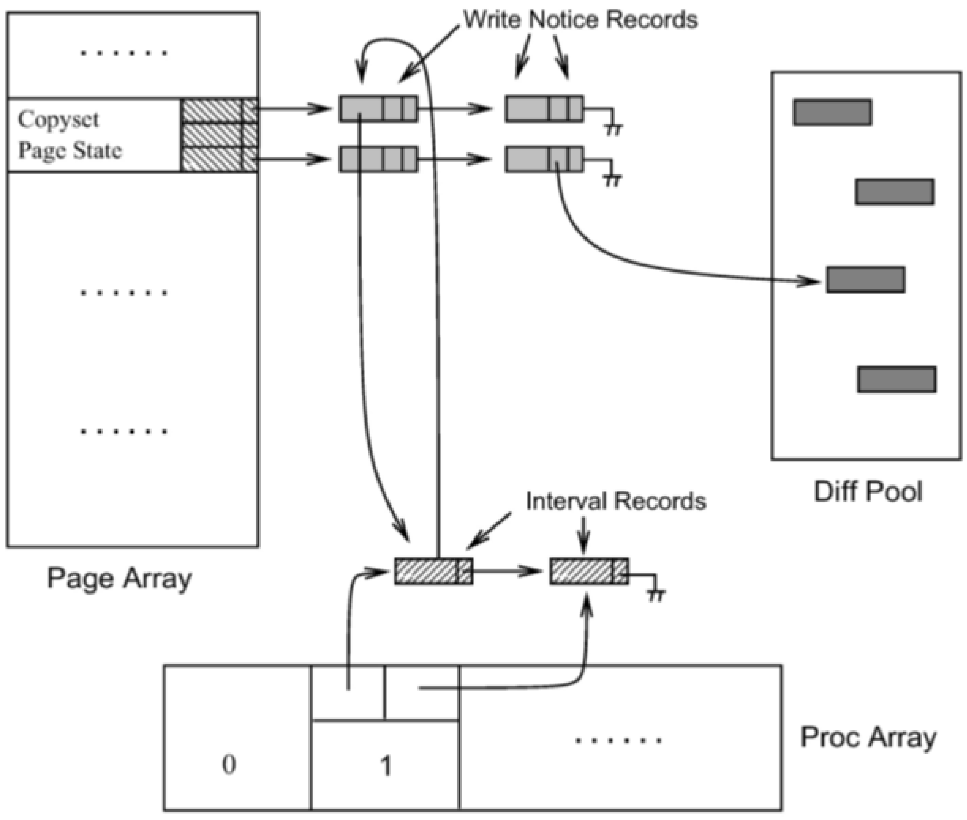

Figure 2: Overview of TreadMarks Data Structures from P. Keleher . *Distributed Shared Memory Using Lazy Release Consistency*, 1994.

Locks

All locks have an assigned manager. Lock management is initially assigned in a round-robin fashion among the processors.

The lock acquire request contains the current vector timestamp of the acquiring processor. The lock request arrives at the processor that either holds the lock or did the last release on it, possible after forwarding by the lock manager. When the lock is released, the release “informs” the acquirer of all intervals between the vector timestamp in the acquirer’s lock request message, and the releaser’s current vector timestamp.

Barriers

Barriers have a centralized manager. At barrier arrival, each client “informs” the barrier manager of its vector timestamp and all of the client’s intervals between the last vector timestamp of the manager that the client is aware of and the client’s current vector timestamp. When the manager arrives at the barrier, it “incorporates” these intervals into its data structures. When all barrier arrival messages have been received, the manager then “informs” all clients of all intervals between their vector timestamp, as received in their barrier arrival message, and the manager’s current vector timestamp. The clients then “incorporate” this information as before. As for locks, incorporating this information invalidates the pages for which write notices were received.

Access Misses

If the faulting processor does not have a copy of the page, it requests a copy from a member of the page’s approximate copyset.

If write notices are present for the page, the faulting processor obtains the missing diffs and applies them to the page. If processor *p* has modified a page during interval *i*, then *p* must have all the diffs of all intervals (including those from processors other than *p*) that have a smaller vector timestamp than *i.*

After the set of necessary diffs and the set of processors to query have been determined, the faulting processor sends out requests for the diffs in parallel. When all necessary diffs have been received, they are applied in increasing vector timestamp order.

Garbage Collection

During garbage collection, each processor validates its copy of every page that it has modified. All other pages, all interval records, all write notice records and all diffs are discarded.

Garbage Collection is triggered when the amount of free space for consistency information drops below a threshold. An attempt is made to make garbage collection coincide with a barrier, since many of the operations are similar.

HIVEMIND APPROACH

Our HiveMind is based on TreadMarks, however changes have to be made in order to accommodate node churn and Golang limitations. Most notably, changes are not guaranteed to be committed into shared memory until a barrier or garbage collection is passed.

Protocol

The TreadMarks implementation by Keleher et al. uses UDP/IP. To simplify development, we will use Go-lang's built in RPC system on top of TCP/IP. This introduces latency but reduces complexity.

Shared Memory Management

To implement the consistency protocol, TreadMarks uses the mprotect system call to control access to shared pages. Any attempt to perform a restricted access on a shared page generates a SIGSEGV signal. The SIGSEGV signal handler examines the local PageArray to determine the page's state. If the local copy is read-only, the handler allocates a page from the pool of free pages and performs a bcopy to create a twin. Finally, the handler upgrades the access rights to the original page and returns. If the local page is invalid, the handler executes the access miss procedure. Go supports the mprotect system call but does not support the ability to install a signal handler for SIGSEGV. Therefore, this mechanism cannot be used by the Go implementation. In order to support this behavior we will need to build a module in Go to mimic this behavior. This module will provide the ability to set access rights of each page and also install handlers to trap on an access violation. The read/write to this memory will be through API calls provided by the module. The read/write will block until the handler has resolved the memory violation similar to the TreadMarks handler.

Inter-Process Communications

The implementation will require a reliable low latency method for passing messages between the processes in the system. The messaging system needs to support a full mesh, i.e. each process is able to talk to every other process in the system. A Go module will be added to support this communication. It will receive the address of the centralized barrier manager to connect to during startup and it will obtain the list of peers and connect to them. It will also implement heartbeats to detect drone failures. It will provide an API to send and receive messages that are used by the rest of the HiveMind implementation.

Centralized Barrier Manager

The first drone to start up is designated the barrier manager. It is responsible for coordinating barriers, and is also the drone contacted to obtain the addresses of the other nodes. In this regard, it is similar to a server. To simplify HiveMind, we assume that the CBM will not fail.

Drone Startup

We seek to implement HiveMind through the use of drones, compiled Go applications utilizing the HiveMind API. We assume that all of the nodes that are used to run the HiveMind DSM can reach each through the IP network, and the nodes IP addresses and ssh credentials are listed in a configuration file. A user logs into first node in the configuration file and launches the drone and passes a configuration file. When the initial drone starts, it calls the HiveMind startup() API to begin the initialization process to allocate a shared memory space and launch the other drones specified in the configuration file. The startup call will use the golang os/exec library to copy the drone executable to the remote node and launch it with the IP address of the CBM.

Manager

Just like in TreadMarks there will be two types of manager drones in our system, lock managers and a centralized barrier manager.

As in TreadMarks, all locks on pages will have a static manager assigned to them, and locks will be given out in a round robin style among drones. When a drone requests to acquire a lock the lock manager will forward the request to the last drone which had released the lock, or the drone which currently holds the lock. The request contains the current timestamp of the drone that is acquiring the lock. Once the lock is released, the releaser drone sends a response to the aquirerer containing for each interval between the acquirer's timestamp and the releasers current timestamp: the process IDs, vector timestamps, and write notices. The aquirerer then uses this information to update it’s data structures and invalids pages which it received a write request for.

In our system memory barriers will be called explicitly; memory barriers will be primarily used in garbage collection in order to discard out of date diffs/write notices/interval records. When a client calls the memory barrier API function it would use the inter-process messaging handler to message to the centralized barrier manager with it’s vector timestamp and a list of all intervals between it’s current timestamp and the manager’s last known timestamp as arguments. The barrier manager will have its own data structures, and it will use this information to update them. Once all clients have messaged the manager, for each client the manager will send back all intervals between the manager’s current timestamp and the client’s current timestamp.

Drone Joining

Due to node churn, new drones can attempt to join HiveMind. The new drone will have to wait until the network’s next barrier or garbage collection. Once a barrier is reached, the network can transfer some of the pages to the new drone (the new drone becomes the manager of those pages).

Drone Failures

Drones can leave the network at any time. Furthermore, there’s no guarantee that the drone is able to “clean up” (i.e. crash or disconnection). In this case, some pages will lose their manager and become inaccessible. The leaving drone may have also held locks for some pages. Hence, managers will have to release the locks of any drone that fail (most likely implemented through some sort of timeout).

In TreadMarks, LRC means that changes to a page is only shared to lock acquirers when the lock is about to be released. However, in TreadMarks these changes take the form of write notices. Therefore on an access miss, if the node we are trying to read diffs from is down, we will use the drone with the next highest interval to get our diffs. In this sense, if a drone has just modified a page and goes down, it’s changes are effectively lost. Thus, it is possible for a drone to acquire a lock, then for the previous node to fail before this node releases, rendering the changes lost. However, TreadMarks (and HiveMind) has the relaxed memory model, where we do not necessarily return the last value written. Hence, changes lost means we return an older value, which still satisfies the memory model.

A failed drone may also be the manager to some pages. Thus, a new manager will have to be elected by the network. This will be done by consensus, a new manager of that page being elected only if all other drones agree that manager is unreachable. While the manager is dead or undecided, the page(s) it held cannot be accessed. Ideally the new manager will be assigned to the drone with the most up to date version of page for that lock (the drone with diffs for the highest interval), breaking ties by drones with the least number of pages that it is managing, finally by comparing IP addresses (or process IDs). However, when drones receive write notice for a given page, they invalidate said page, which means it is possible to enter a situation where no drones currently alive has a copy of the page (page having “disappeared” from the shared memory). To remedy this, when pages are invalidated they are still retained by the drone. The app is not allowed to access the stale copy (since it is invalid), but if no other copies can be found in the network, the stale copy will be resurrected.

Barriers / Garbage Collection

When drones collectively stop work because of encountering barriers or requiring garbage collection, the network can reshuffle pages around. This allows the network to admit new drones and recover from loss of drones.

APPLICATION UTILIZING OUR SYSTEM

To demonstrate that our system is working, we will implement and run applications that demonstrates reading values from and writing values to memory. One application we will implement is QuickSort. To ensure that the program is utilizing memory across multiple drones, we will create an unsorted list that has the length more than the length 2 or fewer drones (including the host of the program) can handle with the allocated memory using HiveMind.

QuickSort (QS) demonstrates the following features of HiveMind:

- Memory allocation: QS starts by list creation (the unsorted list).
- Read: To sort, QS compares value of each cell to a pivot, which is also a value of a cell of the list.
- Write: QS also swaps values of 2 cells after one comparison of values, if needed.

As stated above, QuickSort showcases the basic memory manipulation features of HiveMind. We will also measure the speed up our system gets when compared to a single client version of QuickSort.

ASSUMPTIONS WE WILL MAKE

- Systems will have sufficient memory, potentially to store the entire shared memory
- Drone memory is reliable (no bitrot etc).
- Nodes are trustworthy. This system would typically be deployed to computers in a campus’ lab etc, so we can assume nodes will not cheat
- The centralized barrier manager (ie. the first drone) will not fail
- Drones fail occasionally (not too often). In particular, drones do not fail during recovery (eg. new manager consensus), barriers
- Reliable network (packets do not drop, and they transmit in bounded time)

ASSUMPTIONS WE CANNOT MAKE

- All other drones are reliable
- When or whether drones will reconnect

SWOT Analysis

Strengths	● Group members offer each other support ● Team is willing to take risks
Weaknesses	● Group has trouble making decisions ● Group members have only started learning Go this term ● Only one member has experience using Azure ● TreadMarks is a DSM that does not consider failures, so group is taking a complex system and attempting to add fault-tolerance on top of implementing it ● Golang's inability to be as low-level as C may force compromises
Opportunities	● Azure environment means we can test with more machines than usually possible ● ¾ of us took CPSC 415, and so we have a good idea of how memory works in a simple monolithic OS which could help us here ● We have discussed our work with Ivan's grad students who is knowledgeable in shared memory and distributed systems ● Golang tends to avoid shared memory, instead preferring channels, meaning we are entering poorly charted territories ● TreadMarks uses vector timestamp, possibly enabling easier addition of extra credit objectives (ShiViz, GoVector)
Threats	● Our team does not manage time efficiently ● Other assignments/coursework may interfere with progress on the project. ● Most other implementations of DSM are done at the operating system level, or even the hardware level so there might be unseen obstacles that we have to solve. ● TreadMarks is designed back when multi-core machines are extremely rare. Modern SMP machines are effectively DSMs, so our project may be facing harsh competition against modern CPUs.

TIMELINE

Tasks should be completed by the end of the date specified

March 09	● Group meeting to finalize proposal ● Submission of proposal (Vincent)
March 12	● Skeleton of project files created (Dash)
March 14	● High level design of the virtual memory manager (Dash, Kamil) ● High level design of the inter-processor messaging handler (Vincent, Edward)
March 16	● Implement and unit test virtual memory manager (Dash, Kamil) ● Implement and unit test messaging handler (Vincent, Edward)
March 17	● Data Structures (Dash, Kamil) ○ Implement and Unit test PageArray object ○ Implement and Unit test ProcArray object ○ Implement and Unit test TimeStamp object ○ Implement and Unit test IntervalRecord object ○ Implement and Unit test WriteNotice objects ● Drone Communication (Vincent, Edward) ○ Implement and Unit test remote Drone startup ○ Integrate messaging handler
March 18	● Implement and unit test vector clocks
March 19	● Basic drone implementation ○ Data Structures (Dash, Kamil) ■ Implement and unit test CreateIntervals/Incorporate Intervals ■ Integrate with virtual memory handler ○ Lock Managers (Dash, Kamil) ■ Implement and unit test LockAcquire/Release ○ LRC working between drones (Vincent, Edward) ■ Implement and unit test PageRequest ■ Implement and unit test DiffRequest
March 20	● Centralized barrier manager ○ Implement and unit test barrier (Vincent, Edward) ● Implement and unit test handling access misses (Dash, Kamil)
March 23	● Emailing the assigned TA to schedule a meeting to discuss project status (Edward)
March 24	● Implement and unit test garbage collection (Dash, Kamil) ● Work on surviving failures begins (Team)

March 31	● Debugging (Team) ● Possibly extra credit work, time permitting (TBD)
April 06	● Project code and final reports

EVALUATION

We will deploy our code on Azure. We will run our code on up to eight machines, each configured with Azure's DS1_V2 Standard configuration, which includes 1 vCPU and 3.5 GB of RAM.

TESTING

We will implement a series of unit tests. These unit tests will cover the basic memory manipulating features (Initialize, Read, Write), and also Pointer features (Malloc, Free). Tests will be run when a new feature is implemented or when changes are made, which ensures previously implemented feature/parts are not broken or affected in a negative way by the change(s).

Creation of unit tests will be the duty of the person who implemented the feature or who made the changes.

Integration tests will be developed once smaller components have been developed and are ready to be combined. An example being virtual memory management and the core data structures.

System testing will be done using the QuickSort application. If time permits, more applications such as one involving traveling salesman problems will be implemented to stress test the system.

REFERENCES

1. P. Keleher. *Distributed Shared Memory Using Lazy Release Consistency*.PhD thesis, Rice University December 1994
2. P. Keleher . *Distributed Shared Memory Using Lazy Release Consistency*, 1994.
3. Cristiana Amza,Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations, Computer Volume 29 Issue 2, February 1996, Page 18-28

4. P. Keleher, A. Cox, S. Dwarkadas and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Proceedings of the 1994 Winter USENIX Conference, January 1994
5. Manuel Costa, Paulo Guedes, Manuel Sequeira, Nuno Neves, Miguel Castro: Lightweight Logging for Lazy Release Consistent Distributed Shared Memory, USENIX 2nd Symposium on OS Design and Implementation (OSDI '96), Page 59-74