Checkpointing and Recovery in a transaction-based DSM Operating System

M. Schoettner, S. Frenz, R. Goeckelmann, and P. Schulthess
Ulm University, Department of Distributed Systems
Oberer Eselsberg, 89069 Ulm, Germany
schoettner@informatik.uni-ulm.de

ABSTRACT

Reliability of cluster systems can be improved by periodically saving checkpoints in stable storage. In case of an error a backward error recovery can restart the cluster from the last checkpoint and thus avoiding a fallback to the initial state. Different strategies originally developed for message-passing systems have been adapted for Distributed Shared Memory (DSM) systems. However, it is not sufficient to save only the changed parts of the DSM in a checkpoint. Node-local process contexts must also be included. The latter is hard to realize because the used Operating Systems (OS) are not designed for cluster operation and checkpointing. In this paper we describe the new distributed Plurix OS extending the well-known Single System Image (SSI) concept by storing everything in a DSM, including kernel code and data. Activity is performed using restartable transactions that are synchronized using an optimistic forward validation scheme. We show how these architectural properties simplify the implementation of checkpointing and recovery. Finally, we present measurement numbers that underpin our approach.

KEY WORDS: Reliability, Operating Systems, Distributed Shared Memory.

1. INTRODUCTION

A lot of checkpointing and recovery strategies have been designed and developed for message-passing systems [1]. They range from simple global coordinated algorithms to sophisticated independent techniques.

The results have been adapted for Distributed Shared Memory (DSM) systems starting in 1990 by Fuchs [2]. Numerous subsequent papers discuss the adaptation of checkpointing strategies designed for message-passing systems ranging from global coordinated solutions to independent checkpointing with and without logging [3]. However, the more sophisticated solutions have not been evaluated in real implementations because checkpointing is difficult to achieve in PC-clusters even under global coordination.

The main reason is that if a checkpoint needs to be saved it is not sufficient to save the DSM content but also the local process context needs to be observed. Otherwise errors may occur during recovery, e.g. unsaved and not restored kernel locks. This is not a trivial task because the architecture of the underlying Operating System (OS), typically a Unix system or Microsoft Windows, has not been designed for cluster operation and checkpointing. As a consequence some projects, e.g. Kerrighed modify the OS kernel to alleviate these problems [4]. Nevertheless, there are still limitations and there is a considerable overhead during checkpointing.

Saving and restoring kernel contexts and local resources is also an issue in process migration facilities. Mosix developed for Linux leaves a process fragment on the old machine behind to access non-migratable resources like files and locks [5]. The migrated process on the target machine communicates with its successor using RPC.

In this paper we describe our new distributed Plurix OS specifically tailored for cluster operation which avoids these difficulties by extending the well-known Single System Image (SSI) concept. Plurix is the first DSM system storing everything within shared memory including kernel data, code, even all drivers. As a consequence checkpointing is simplified.

Plurix has introduced a strong consistency model, called transactional consistency based on restartable transactions and an optimistic synchronization scheme [6]. Supporting restartability by the OS and even at the kernel level simplifies the recovery procedure.

Orthogonal persistence [7] is another DSM property in the sense that any object reachable from the root of the cluster-wide name service can persist independent of its type. Persistence is directly supported by our checkpointing and recovery facilities meanwhile making de- and serialization functions required for file-based systems superfluous.

In the following section we shortly review the important properties of the Plurix DSM. In the third section, we present our checkpointing facility. Subsequently, we describe the error detection and recovery strategy. In the fifth section we evaluate our implementation. Finally, we compare our system to related work and conclude.

2. THE PLURIX ENVIRONMENT

Memory Management

After the first Distributed Shared Memory (DSM) prototype IVY has been presented in 1988 a lot of systems have been developed to mainly support parallel algorithms. Most of the approaches were built on top of existing Operating Systems (OS), like Unix, Microsoft Windows, or using the Mach kernel.

The Plurix project implements a distributed OS for PC clusters customized for page-based DSM operation. The Memory Management Uni (MMU) detects non-local memory accesses and the kernel retrieves the requested page from the cluster. Current address space limitations will vanish with 64 Bit processors like Intel IA64.

Page-based DSM systems are faced with the problem of false sharing causing page trashing. False sharing occurs if at least two objects reside on a memory page and within a certain time interval each of the object is accessed by a different node access conflicts may arise which are semantically unnecessary. Our system alleviates this serious problem by a concurrent object relocation facility based on a book keeping of references [8]. The latter is possible because the kernel is able to identifiy objects residing on an affected page and thus relocating one or several objects to other pages.

In traditional DSM systems programmers must use special functions for allocating data in DSM memory. The Plurix DSM is managed as a heap and accessed like local memory. The benefits of a heap organization have also been identified in Murks [9] but Plurix goes one step beyond by also storing code and runtime structures in the DSM.

Single-System-Image (SSI) computing architectures have been the mainstay of high performance computing for many years. In a system implementing the SSI concept, each user gains a global and uniform view on available resources and programs and provides the same libraries and services on each node in the cluster, which is very important for load balancing and migration of processes. We extend the SSI concept by storing OS, kernel, and all drivers in the DSM [10].

Heap management must be designed carefully to avoid unnecessary false-sharing situations, e.g. it is recommended that each node allocate memory in a different part of the DSM avoiding interferences. Furthermore, vital classes of the system need special protection, e.g. the page-fault handler. We have introduced different memory pools to support these requirements.

Distributed garbage collection relieves programmers from explicit memory management. Unreferenced objects can be collected very easily using the compiler-supported bookkeeping of references, [8].

Because all nodes of a cluster operate concurrently on the DSM adequate synchronization mechanisms are required described in the following subsection.

Transactional Consistency

The DSM community has proposed a lot of memory consistency models especially weaker ones to improve efficiency [11]. Because weaker consistency models are hard to program we have introduced a strong consistency model called *transactional consistency*.

Memory pages are distributed and read-only copies are replicated in the cluster. When writing to a memory page all read-only copies are invalidated and the writing node becomes the new owner of that page. Inconsistencies are avoided by synchronizing memory accesses from different nodes using our transactional consistency model [6].

In contrast to existing memory consistency models we do not synchronize memory after each memory write access but bundle several operations within a transaction (TA). In case of a conflict between two transactions we rely on the ability to reset changes made by a TA. This conflict resolution scheme is known in the database world as optimistic concurrency control. Optimistic concurrency control occurs in three steps: the first step is to monitor the memory access pattern of a TA. For this purpose we use the built-in facilities of the MMU.

The next step is to preserve the old state of memory pages before modifications. Shadow images are created, saving the page state previous to the first write operation within a TA. These shadow pages are used to restore the memory in case of a collision, as described in the next step.

During the validation phase of a terminating TA the access patterns of all concurrent TAs in the cluster are compared. In case of a conflict the TA is rolled back using the shadow pages otherwise the latter are discarded.

Currently, we have implemented a first-wins collision resolution basing on a circulating token. Only the current owner of the token is allowed to commit. During a commit the write-set of the TA is sent as a broadcast to all nodes in the Fast-Ethernet LAN. All nodes in the cluster compare the write set with their running TA to detect conflicts and to abort voluntarily. In future we plan to integrate other conflict resolution strategies to improve fairness.

Scheduler

Instead of having traditional processes and threads the scheduler in Plurix works with transactions. We have adopted the cooperative multitasking model from the Oberon system, [12]. In each station there is a central loop (the scheduler) executing a number of registered transactions with different priorities. Any TA can register further transactions. System TAs, e.g. the garbage collector are automatically registered by the OS. Furthermore, the OS automatically encapsulates all user commands within a transaction.

Transactions should be short to minimize collision probability and it is the task of the programmer to split up long running transactions.

Name Service

Any heap object can be registered in a cluster-wide name service and is later retrieved via directories and subdirectories. This corresponds to the directory structure of traditional file systems but the functionality of the name service is extended to include scoping in the Java compiler, to store configuration information, and to cover all naming issues occurring in the OS.

The Plurix compiler automatically registers symbol information during the compilation in the name service. Methods of classes or instances may be immediately invoked using this persistent symbol information.

Any heap object reachable from the name service root is not garbage. Checkpointing and Recovery for the DSM implement persistence for all objects in the heap. These properties are known as orthogonal persistence - any class, instance or array may persist.

3. CHECKPOINTING

State of the art PC-Clusters are built using Linux or Microsoft Windows but implementing checkpointing and recovery in these OSs is difficult because it is not sufficient to save the DSM content but also the local process context needs to be observed. The latter includes internal kernel data structures, open files, used sockets, pending page requests, ... which can be read only at kernel level. Resetting the kernel and process context in case of a rollback is also challenging because of the complex OS architectures. As a consequence taking a checkpoint is time consuming and checkpointing intervals are typically large, e.g. 15-120 min. for the IBM LoadLeveler.

By extending the well-known Single System Image (SSI) concept we avoid these architectural drawbacks [10]. Storing the kernel and its data in the DSM makes it easy to save all required data. Furthermore, rollback in case of an error is not very difficult in Plurix because the OS and all its applications are designed to be restartable anyway.

Our first prototype uses a central PageServer for storing checkpoints - consistent heap images in an incremental fashion on hard disc. Between two checkpoints the PageServer uses a bus snooping protocol to intercept transmitted and invalidated memory pages to reduce the amount of data to be retrieved from the cluster at the next checkpoint. These snooped pages are stored in the so-called `snooped_pages_buffer` located outside the DSM. This buffer stores a fixed number of pages (e.g. 512) and is not used to serve for page requests.

If a checkpoint must be saved the PageServer collects invalidated pages from the cluster that have not been snooped since the last checkpoint or if a entry in the `snooped_pages_buffer` is invalid. The latter occurs if a page is transmitted over the network once, changed and invalidated by a node, and then no more exchanged over the network.

Currently, all memory pages are written to disk synchronously. During this short period transactions may be executed on each node but cannot commit.

Linear-Segment Structure

Commodity hard discs achieve write rates up to 45 MB/s when writing subsequent sectors. This write-performance breaks down to below 1 MB/s when writing scattered sectors requiring a lot of head positioning.

With respect to this property we have developed the linear-segment writing technique for the PageServer. We use no file-system but write raw sectors using an own UDMA IDE disc driver. Beyond writing most times in a linear fashion we also ensure that written data will never get lost also in case of a PageServer failure.

The disc partition is divided in blocks each consuming up to 161 sectors, see figure 1. The number of pages per block can be adjusted statically. The first sector contains a header with 24 Byte for the block and for each stored page thus supporting up to 20 pages per block (24 + 20*24 = 504 bytes < 512 bytes). Subsequently, the following sectors store the content of up to 20 pages each 4 KB, which comes to 161 sectors per block.

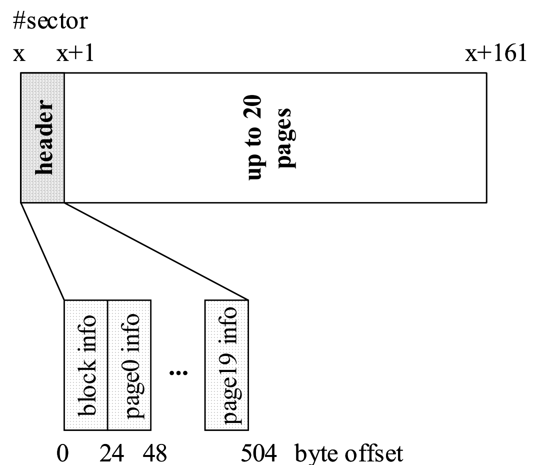

Figure 1, Block structure

The `block_info` contains a 64-Bit timestamp identifying the checkpoint time the block was written. Additionally, there is a sector ID and some reserved flags, e.g. “last block of checkpoint”.

The `page_info` stores information for each page within the block: “page number (64-Bit, currently only 32-Bit used) and a last change timestamp (commit number of TA that modified that page the last time).

Meta Data for Checkpoint Management

The PageServer tracks pages changed since the last checkpoint in a buffer called `DSM_info` consuming 31 MB with 38 Byte entries for each page in the DSM. The buffer is stored always at the beginning of the disk to ensure linear writing with maximum speed.

The `DSM_info` meta-data is updated on disk with each checkpoint. When the PageServer crashes during this operation the data can be reconstructed by reading through the disk. Of course the latter is slow but should occur very seldom. Optionally, a `last_DSM_info` could also be maintained on disk after the actual meta-data to alleviate this special error. During the shutdown of the PageServer a checkpoint is written to disk which includes also a new version of the `DSM_info` buffer.

The latter is also cached in main memory to speed up read and write access. An entry in the `DSM_info` buffer consumes 38 Byte:

- `last_changed` (64 Bit): commit number of the last TA that has written this page. Updated using the write-sets of committing TAs.
- `last_seen` (64 Bit): commit number of the PageServer when it has seen the page on the network. If `last_seen` is less than `last_changed` the page in the `snooped_pages_buffer` is invalid and needs to be retrieved from the cluster when saving a new checkpoint. Because the `snooped_pages_buffer` is not stored in the DSM pages must be invalidated by the PageServer manually.
- `last_saved` (64 Bit): commit number of the page on disc. If `last_saved` is less than `last_changed` the page must be written to disc with the next checkpoint.
- `last_SPos` (64 Bit): the lower 32 Bits store the block number of the most recent version of the page. The upper 32 Bits are used to store the offset within the block.
- `Loc` (32 Bit): defines the location of the page:
    - NOT_HERE: page is neither in the local DSM (locally replicated pages) nor in the `snooped_pages_buffer`.
    - OWNER: Shows if the PageServer is the owner of a page.
    - LOCAL_DSM: If a page is present this bit shows if the page is in the local DSM or in `snooped_pages_buffer`.
    - ADDRESS: Defines the page frame within the `snooped_pages_buffer`. This value is only valid when the page is stored in the `snooped_pages_buffer`.
- `PFlags` (16 Bit): page flags of the kernel.

Page Requests to the PageServer

Page requests are only processed by the owner of a page, never by nodes having read-only replicates. The owner of a page can change dynamically in Plurix by writing to a page. With respect to page requests the PageServer acts like a normal node replying only if it is the owner of the requested page.

The PageServer has never the ownership for pages stored in the `snooped_pages_buffer`.

The PageSever decides if it must process a page request using the information stored in the `DSM_info` buffer. If `last_seen` > `last_changed` the page is:

- only in main mem.: if `last_saved` < `last_changed`
- may be on disc: if `last_saved` > `last_changed`

If `last_seen` < `last_changed` another node should serve this page request. If the request is sent multiple times the PageServer assumes that a serious error occurred. A node has changed a page and crashed before the PageServer saved this page. Thus the PageServer must start the recovery procedure.

4. ERROR DETECTION & RECOVERY

Network Packet Loss

Currently, we support clusters running within a single Fast Ethernet LAN segment and assume a fail-stop behavior of nodes. Most DSM systems typically use a reliable multicast or broadcast facility to avoid inconsistencies caused by lost network packets. Because of the low error probability of a Hub-based LAN we are not willing to impose the overhead by a reliable communication during normal operation. Instead we rely on a fast error detection, fast recovery, and the quick boot option of our cluster OS. A node in a Plurix cluster can perform a cold start within 250ms.

As described in section 2 we have implemented transactional consistency where committing transactions are serialized using a token. We can easily introduce a logical global time (a 64-Bit value) incremented each time a TA commits. In the latter case the new time is broadcasted to the cluster and each node updates its time variable. A node can immediately detect if it missed a commit and ask for recovery.

If the commit message cannot be sent with one Ethernet frame, the commit number is incremented for each commit network packet. Thus we avoid inconsistencies if a node did miss one packet of a multiple packet commit.

Furthermore, any page or token request always includes the global time value of the requesting node. If such a request contains an out-of-date commit number it is not processed but recovery is started. Thus a node that missed a commit packet is not able to commit a transaction that used invalid data because it is not granted the token.

Running the cluster in a switched LAN increases packet loss probability because the switch may discard packets in case of a high network load. For such a scenario we cache a small history of TAs (backup copies of changed pages) on each node. This information can be used to synchronize nodes that missed a commit message without a cluster reset.

Node Failure

If a single node fails temporarily it can reboot and join the DSM again. If the PageServer detects missing pages during the next checkpoint that were lost because of the node failure the cluster is reset to the last checkpoint. The same works if multiple nodes fail temporarily or permanently.

If a page request of node is not serviced it retries the request three times and after that it assumes that the page got lost and requests recovery.

PageServer Failure

If the PageServer fails temporarily it will detect a running cluster during its reboot and will start a cluster reset from its last checkpoint on disk. Currently, we have only a single PageServer with the problem when failing permanently crashing the cluster. In future we plan to substitute the PageServer by a distributed solution with data replication so we can also tolerate multiple PageServer failures.

Network Partitioning

The network might be partitioned temporarily into two or more segments. Only one token and one PageServer is available in any of these segments. Nodes within each segment send page and token requests. If either request cannot be satisfied the segment tries to recover by contacting the PageServer. Only the segment with the PageServer can recover the others have to wait until the PageServer becomes available again.

In the worst case more than one segment can commit TAs and therewith make changes in the DSM. When the segments melt again the error can be detected if global time is different in the segments. In the case that two or more segments have made changes and have the same global time the DSM might be in an inconsistent state that is not detected. Numerous strategies have been proposed in literature to solve this problem all discarding changes of one partition. Because Plurix stores everything within the DSM we hope to develop an strategy melting heaps using symbolic information about objects.

Resetting Nodes

As all OS context information is stored in the DSM and nodes can reboot in about 250ms we prefer to reboot the cluster in case of falling back to the last checkpoint. Enforcing a redraw of the desktop and all its windows.

If nodes are rebooted device states get lost. Applications depending on the latter must be able to restore on demand. For example textures stored in graphic card memory must be downloaded during the next redraw.

5. EVALUATION

Former measurements with a distributed multi-player game (without checkpointing) have shown that the token and page latency for our DSM is around 120µs and 526µs respectively with a four-node cluster in a Fast Ethernet cluster [13].

The measurements in this paper are carried out on a cluster with four PCs interconnected by Fast Ethernet Hub. Each node is equipped AMD Athlon 2,5 XP CPU and 512 MB RAM. To evaluate our checkpointing implementation we run concurrently a ray tracer.

The latter has been adapted from an MIT course [14] to our transactional DSM system. The ray tracer is executed on each node except the PageServer. The ray tracer computes the image shown in figure 2 in ~530s. The image has 2560x2048 pixels and 32-Bit color depth.

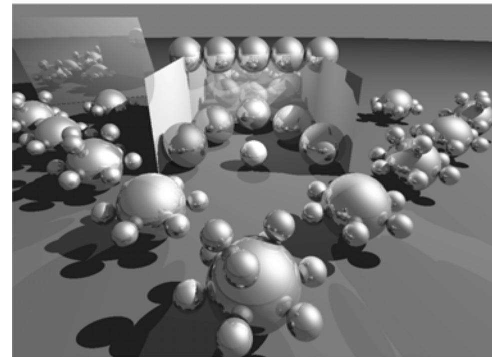

Figure 2, The test image

During the computation the PageServer periodically writes checkpoints to hard disk. During this phase transactions may execute on nodes but cannot commit until the PageServer has finished.

The following figure 3 shows the number of pages that the PageServer must request from the cluster and write to disk depending on the checkpointing frequency. Furthermore the time needed to finish a checkpoint is also shown.

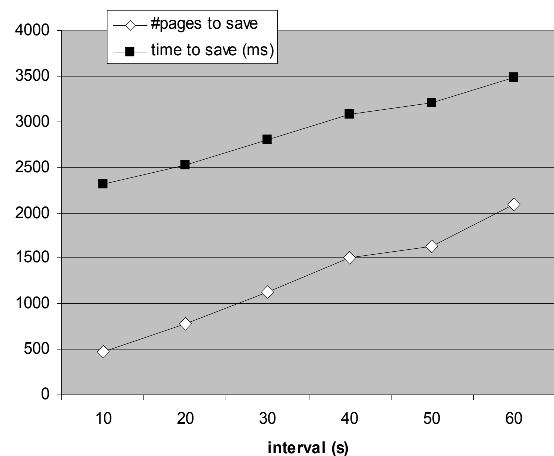

Figure 3, Checkpointing time and #pages to save

This number of pages to save includes not only pages changed by the ray tracer but also those modified by the OS executed on each node. Of course the amount of modified data grows with the checkpoint interval from 2 MB to 8 MB caused by the progressing ray tracer.

The time required to save a checkpoint grows only moderately from 2,3s to 3,5s. The time required to save the meta-data described in section 3 consumes always ~1s. The remaining time depends on the number modified pages since the last checkpoint.

Writing memory pages synchronously to disk causes of course most of the time compared to fetching them over the network. Nevertheless, the current checkpointing implementation is comparable fast because it includes all contexts including changes made by the OS.

6. RELATED WORK

Typically, DSM systems offer special memory allocation functions and it is the task of each application to decide which data to allocate in DSM and what to store in normal node-local memory. The main motivation for this approach is to avoid consistency management for data that is not shared. But it is natural that there are dependencies between data items stored inside and outside the DSM.

Numerous checkpointing strategies have been suggested for DSM systems in the literature [3]. However, these papers are mostly limited to the efficient checkpointing and recovery of the DSM and do not solve the problems resulting from the mentioned dependencies. Typically, it is the task of each application to respect the limitations or to act itself properly in case of a recovery.

The most close to our approach is the Kerrighed project adapting a Linux kernel for DSM operation including checkpointing and recovery [4]. The system is still under development and also implements a global coordinated checkpointing including process contexts in each checkpoint stored on disk. The latter must be saved separately on each nodes disk. The global checkpoint is finished if each node has written its checkpoint data to disk. Depending on the node-local data to be saved this may consume considerable time. Furthermore saving and restoring kernel contexts is complex because the Linux architecture is not designed for cluster operation and backward error recovery. As a result there are limitations what contexts outside the DSM can be checkpointed and recovered.

7. CONCLUSIONS AND FUTURE WORK

We believe that a persistent DSM offers new perspectives for distributed programs, e.g. multi-player games, virtual worlds, etc. Transactional consistency offers strong consistency at reasonable performance thus simplifying distributed programming. Storing all data and code in the DSM including kernel data simplifies checkpointing in contrast to traditional cluster OSs.

Furthermore, restartability is directly supported by the OS by restartable transactions and an optimistic synchronization scheme. Thus resetting the cluster in case of an error to the last checkpoint is not difficult. Even hardware contexts can be reset very fast because the OS can performa a cold reboot in about 250ms.

In future work we plan to develop more sophisticated checkpointing strategies based on a distributed Page-Server writing asynchronously to disk.

8. REFERENCES

[1] M. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson, “A Survey of Rollback-Protocols in Message-Passing Systems”, *Technical Report CMU-CS-99-148*, Carnegie Mellon University, Pittsburgh, USA, 1999.

[2] Kun-Lung Wu and W. Kent Fuchs, „Recoverable Distributed Shared Virtual Memory”, in: *IEEE Transactions on Computers*, 39(4):460-469, April 1990.

[3] C. Morin and I. Puaut, “A Survey of Recoverable Distributed Shared Virtual Memory Systems”, in: *IEEE Transactions on Parallel and Distributed Systems*, Vol. 8, No. 9, Sep. 1997.

[4] http://www.kerrighed.rog

[5] http://www.mosix.org

[6] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess, “Optimistic Synchronization and Transactional Consistency”, in: *Proc. of the Workshop on Distributed Shared Memory on Clusters, IEEE Intl. Symposium on Cluster Computing and the Grid*, Berlin, Germany, 2002.

[7] M. J. Jordan, T. Prinzezis, M. P. Atkinson, L. Daynes, and S. Spence, "An Orthogonally Persistent Java", *SIGMOD Record*, Volume 25, 1996.

[8] R. Goeckelmann, S. Frenz, M. Schoettner, P. Schulthess, “Compiler Support for Reference Tracking in a type-safe DSM”, in: *Proc. of the Joint Modular Languages Conference*, Klagenfurt, Austria, 2003.

[9] Markus Pizka and Christian Rehn, “Murks - a POSIX threads based DSM system”, in: *Proc. of the Intl. Conference on Parallel and Distributed Computing Systems,* Anaheim, USA, 2001.

[10] R. Goeckelmann, M. Schoettner, S. Frenz, P. Schulthess, “A Kernel Running in a DSM - Design Aspects of a Distributed Operating System”, to appear in: *Proc. of the IEEE International Conference on Cluster Computing*, Hong Kong, 2003.

[11] D. Mosberger, “Memory Consistency Models”; *Technical Report TR93/11*, Department of Computer Science, University of Arizona, 1993.

[12] N. Wirt and J. Gutknecht: *„Project Oberon“*, ACM Press, Addison-Wesley New York 1992.

[13] M. Schoettner, M. Wende, R. Goeckelmann, T. Bindhammer, U. Schmid, P. Schulthess , “A Gaming Framework for a Transactional DSM System”, in: *Proc. of the Workshop on Distributed Shared Memory on Clusters, IEEE Intl. Symposium on Cluster Computing and the Grid*, Tokyo, Japan, 2003.

[14] http://graphics.lcs.mit.edu/classes/6.837/F01