

# Fault Tolerance in a DSM Cluster Operating System

Michael Schoettner, Stefan Frenz, Ralph Goeckelmann, Peter Schulthess

Distributed Systems  
University of Ulm  
Informatik o-27  
89069 Ulm

[schoettner|frenz|goeckelmann|schulthess]@informatik.uni-ulm.de

**Abstract:** Plurix is a new distributed operating system for PC clusters. Nodes communicate using a Distributed Shared Memory (DSM) that stores data and code, including drivers and kernel contexts. This approach simplifies global resource management and checkpointing. Instead of executing processes and threads our operating system uses restartable transactions combined with an optimistic synchronization scheme. Beyond restartable Transactions the cluster itself is restartable allowing a simple and fast reset of the cluster in case of a non local error. In this paper we describe relevant parts of the Plurix architecture and discuss how these properties simplify checkpointing and recovery. We also present our current approach for error detection, checkpointing, and recovery.

## 1 Introduction

Expensive super computers are more and more replaced by cheap commodity PCs clusters. They aggregate resources like CPU and memory thus enabling the execution of parallel applications. But also the inherent redundancy of hardware and data replication makes clusters interesting for high availability services.

Implementing a Single System Image (SSI) is a elegant approach to simplify global resource management in a cluster. To avoid mechanism redundancy, conflicting decisions and decrease the software complexity, resource management should be organized in a unified and integrated way. We believe it is natural that this is the task of a distributed operating system (OS).

We have developed the Plurix distributed OS using the proven concept of Distributed Shared Memory (DSM) to simplify data sharing and consistency management. Plurix extends the traditional SSI concept by storing everything in the DSM, including kernel code and all data [Go03].

Reliability of cluster systems can be improved by periodically saving checkpoints in stable storage. In case of an error a backward error recovery restarts the cluster from the last checkpoint and avoids fallback to the initial state.

Numerous checkpointing strategies have been proposed for message-passing systems and several of them have been adapted to Reliable DSM (RDSM) systems. However, it is not sufficient to save only the modified parts of the DSM within a checkpoint but also node-local process contexts must be included. The latter is hard to realize because the OSs typically used, e.g. Linux are not designed for cluster operation and checkpointing. By storing almost everything within the DSM this task is simplified in Plurix.

After the occurrence of an error a checkpointing system falls back to the last checkpoint. Resetting execution to the last checkpoint requires also resetting all local OS states to the last checkpoint which is not trivial in traditional OSs.

Instead of using processes and threads Plurix executes transactions that are synchronized using an optimistic strategy. In case of a collision between overlapping transactions at least one transaction is aborted and may be automatically restarted. Because restartability is a fundamental property of the Plurix OS and because nodes can be restarted in about 250ms recovery becomes simpler and faster. Nevertheless, device contexts stored outside the DSM are difficult to reset and possible strategies under study. Basic architectural properties of the Plurix OS appear to offer interesting perspectives for a fault tolerant cluster.

In the following section we present properties of the Plurix architecture relevant for checkpointing and recovery. In section three we present our current checkpointing implementation. Subsequently, we show error detection schemes and recovery procedures for our system. In section five we give some measurement figures and an outlook on future work.

## **2 The Plurix Operating System**

The Plurix OS is a Java-based implementation designed for PC clusters (min. i486). We have a test bed consisting of 16 Athlon XP 2,2 nodes, 512 MB RAM, IDE Disk, connected by hub-based or switched Fast Ethernet. Applications, kernel, and drivers are implemented in Java with our own Java compiler offering a few minor language extensions to support hardware-level programming [Sch02].

### **2.1 Memory Management**

After the first Distributed Shared Memory (DSM) prototype IVY [Li88] has been presented many DSM systems were developed to support parallel applications. Most prototypes were built on top of existing operating systems (OS).

The Plurix project implements a new distributed OS for IA32 clusters customized for page-based DSM operation. Numerous page-based DSM systems have been developed, e.g. IVY, Treadmarks [Am94], SODA [Co96], all of them facing the problem of false sharing causing page trashing. Plurix alleviates this problem by an incremental object relocation facility based on a book keeping of references [Goe03].

In traditional DSM systems programmers must use special functions to allocate memory in DSM. In Plurix the DSM is managed as a heap and accessed like local memory. Furthermore, Plurix stores even code, kernel and driver contexts in the DSM heap.

Heap management is carefully designed to avoid unnecessary false-sharing situations, e.g. it is recommended that each node allocates memory in a different part of the DSM to avoid interferences. Instead of statically dividing the address space into fixed size parts we have developed smart memory allocators dynamically allocating a larger block used for subsequent allocations [Goe02]. A few vital classes of the system need special protection (e.g. the page fault handler) and are stored in a special memory pool [Goe02].

Distributed garbage collection relieves programmers from explicit memory management. Unreferenced objects are easily collected using the compiler-supported bookkeeping of references [Goe03]. In a DSM environment it is not a good idea for a distributed garbage collection (GC) scheme to mark heap blocks during a full traversal because of the expensive network traffic and the growing collision probability with other nodes.

Because all nodes of a Plurix cluster operate concurrently on the DSM adequate synchronization mechanisms are required to preserve memory consistency.

## 2.2 Transactional Consistency

The DSM community has proposed numerous memory consistency models especially weaker ones to improve efficiency [Mos93]. Because we store kernel and driver contexts within the DSM we require a strong consistency model that might be extended by weaker ones for optimization of specific algorithms. Our so called *transactional consistency* uses restartable transactions and an optimistic synchronization scheme [We02]. 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.

In contrast to existing strong memory consistency models we do not synchronize memory after each memory write access but bundle millions of 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.

The latter 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 Memory Management Unit (MMU). The next step is to preserve the old state of memory pages before modifications. Shadow images are created, saving the page state prior to the first write operation within a TA. These shadow pages are used to restore the memory in case of a collision. 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 arbitrating 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 broadcast to all nodes in the Fast-Ethernet LAN. The cluster nodes compare the write set with their running TA to detect conflicts and to abort voluntarily. In future we plan to integrate alternative conflict resolution strategies to improve fairness.

### **2.3 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, [WG92]. In each station there is a central loop (the scheduler) executing a number of registered transactions with different priorities.

The garbage collector is a pre-registered TA called either dynamically or on demand. Any TA can register additional transactions. Likewise the OS automatically encapsulates commands within a transaction. Transactions should be short to minimize collision likelihood. The lean design of the OS and of its components ensure short transactions (~ 1 second), e.g. our bootstrapped Java compiler compiles 10.000 lines per second. But of course long running computations may be split up by the programmer into several TAs.

### **2.4 Data stored outside Distributed Shared Memory**

Almost all data is stored within the DSM except shadow pages, page tables used by the MMU, and a node number. The latter is used to select a station object in the DSM that is the anchor for node-local instances, e.g. memory manager, drivers etc.

Input and output operations need special consideration with respect to restartability of transactions. Input events from the hardware must not get lost in case of a transaction abort, e.g. the user does not want to retype a key. On the other side output of transactions should not be duplicated in case of a restart. This gap between non-restartable interrupt space and DSM space is bridged by our *smart buffers*. The read and write pointers are stored in the DSM but the buffer itself not. For more details refer to [Bin02].

## **3 Checkpointing**

Many checkpointing and recovery strategies have been designed for message-passing systems [EI99]. They range from simple global coordinated algorithms to sophisticated independent techniques. Results have been adapted for DSM systems starting in 1990 by Fuchs [WF90]. 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 [MP97]. However, the more sophisticated solutions have not been evaluated in real DSM implementations because checkpointing is difficult to achieve in PC-clusters even under global coordination.

The main difficulty is that when a checkpoint is 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 OS, typically a UNIX system or Microsoft Windows, has not been designed for cluster operation and checkpointing. As a consequence some projects, e.g. Mosix [Mos], Kerrighed [Kerr] modify the OS kernel to alleviate these problems. Nevertheless, there are still limitations and there is a considerable overhead during checkpointing.

Resetting the kernel and process context in case of a roll back is also challenging because of the complex OS architectures. As a consequence taking a checkpoint is time consuming, e.g. 15-120 min. for the IBM Load Leveler [Kan01].

### 3.1 PageServer

By extending the well-known Single System Image (SSI) concept we avoid these architectural drawbacks [Go03]. 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 quickly done in Plurix because the OS can reboot in 250ms and if a reboot is not necessary the OS is designed to be restartable anyway because of the transaction-based processing.

Currently, our first prototype uses a central PageServer inside the DSM for storing checkpoints – consistent heap images, as all required data resides in the DSM. Between two checkpoints the PageServer snoops the Ethernet in promiscuous mode to intercept invalidated and transmitted pages to possibly reduce the amount of data to be retrieved from the cluster at the next checkpoint. These snooped pages are stored in the so-called snooping buffer with a fixed number of pages (e.g. 512).

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 snooped page is invalid. The latter occurs if a page is transmitted over the network once, changed and invalidated by a node, and then notexchanged again.

Currently, all tables of the PageServer and all changed memory pages of the DSM are written to disk synchronously resulting in about 2.5 seconds for saving a checkpoint when executing a parallel ray tracer in the cluster and a checkpointing interval of 10 seconds. During the finalization of a checkpoint transactions may be executed on each node but cannot commit.

In future we will introduce optimizations for the checkpointing strategy. An independent PageServer which does not actively participate in the DSM avoids reloading tables in case of a cluster restart as the PageServer itself doesn't depend on a potentially corrupted heap. Thus it is possible writing asynchronously to disk which enables concurrent requesting and saving of pages, especially during idle of the PageServer or the network. As the time required to collect outstanding pages is reduced substantially and as it is not necessary to write all tables to disk, the period of monopolization the commit token while finalizing a checkpoint is reduced to a fraction of a second.

Additionally an integrated distributed PageServer saving checkpoints asynchronously will be examined. Storing pages in a redundant fashion on different PageServers will also tolerate permanent PageServer failures. Instead of writing checkpoints to disk they can be saved in volatile memory and swapped out in the background. This approach requires that always several copies of a checkpointed memory page exist in the cluster. Storing checkpoints in volatile memory has been discussed by Morin [Cab95].

### **3.2 Linear Segment**

Currently, data is stored in an incremental fashion using the linear segment strategy [Fre02]. This algorithm unleashes the full potential of modern hard disks by writing mostly on consecutive sectors. Several pages are combined to a single block together with information about page number, transaction number of last modification and some additional recovery data. On the one hand these linear segment blocks provide necessary information in case of unlikely PageServer errors and on the other hand they speed up transfers to disk by writing larger units.

After a wrap-around of the hard disk volume, both internal tables (used during normal operation) and meta data inside the block (used in case of recovery) provide information to decide which block is still in use. If not all pages inside a block are active, a group of blocks with few required pages may be merged to create unoccupied blocks. This reorganization is not mandatory if another block is free, but can be done on the fly and helps to keep up the write performance. As the hard disk should be much larger than the addressable DSM, there are always unused pages to de-allocate. An alternative to reorganization would consider the time of the last modification: the frequency of write access differs on each page, especially on code pages there is usually no write access. Thus pages can be categorized as cold, warm and hot pages. After a wrap-around the position on the hard disk is at the oldest block, containing mostly obsolete pages. If there are still used pages, they are very cold, so further improvement would be collecting these cold pages to another disk or partition, as these pages contain very rarely changed data, for example code.

## **4 Error Detection and Recovery**

### **4.1 Network Packet Loss**

Currently, we support clusters running within a single Fast Ethernet LAN segment and assume fail-stop behaviour of nodes. 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 from an acknowledged communication during normal operation. Instead we rely on a fast error detection, fast recovery, and the quick boot option of our cluster OS.

As described in section two we have implemented transactional consistency where committing actions from transactions are serialized using a token. We introduce a logical global time (a 64-Bit value) incremented each time a TA commits. The new time is then broadcast to the cluster and each node updates its logical time. A node can immediately detect that it must have missed a commit and it will ask for retransmission of missing write sets (see 4.2) or for recovery action.

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. As a consequence a node that missed a commit packet is not able to commit a transaction that introduces invalid data because it is never granted the token.

Running the cluster in a switched Ethernet LAN increases packet loss probability because the switch may discard packets in case of high network load. For such network environments we cache a small history of TAs on each node that can be used to synchronize nodes that missed a commit message thus avoiding a cluster reset.

#### **4.2 Transaction History Buffer**

To memorize the TA history it is not necessary to keep a backup copy of the data of modified pages in the history buffer. The addresses (page number) of the modified pages are sufficient. To recover the write set of a specific TA all page numbers which have been sent in the original write set must be determined. Therefore the buffer should contain pairs of page number and associated commit number. As this would lead to a large buffer, due to the 64-bit commit number, the buffer is split into two parts, the page address buffer (PAB) and the commit number buffer (CNB). The PAB only contains page numbers without the corresponding commit number but continuously stored for each TA. The CNB contains the commit number and the start and end position of the associated pages in the PAB.

To ensure that any size of TA can be remembered, the PAB must be large enough to contain the page numbers of all DSM pages. In current 32-bit architectures this leads to a buffer size of 4 MB. As this buffer would be overwritten during the next commit phase, the buffer needs to be twice the maximum size of modified pages. For 64-bit architectures it can not be assured that each TA can be stored in the history buffer due to the huge amount of possible modified pages. If in such an architecture the number of modified pages exceeding a threshold, the TA is not stored in the history buffer and the PageServer is used to assure DSM persistency.

4 KB of memory are sufficient for the CNB, since each entry consists of the 64-bit commit number and two 32-bit offsets into the PAB, which permits to buffer for 256 TAs.

Each node contains its own TA history buffer which is meaningless to all other nodes. Storing the latter in the DSM will waste DSM address space without providing any advantages. Hence it is reasonable to store the buffer in local memory. Additionally, the buffer is not saved by the PageServer as the information included in it is obsolete after a cluster restart.

In case of an out-of-date commit number of a node, all transactions on this node are stopped. The node starts the recovery procedure by sending a special message including its current commit number to all other nodes. The node which has the write set with the next higher commit number in its TA history buffer resends the write set to the requesting node and the lost commit can be recovered. If no valid entry can be found in any TA history buffer the cluster must be reset to the last checkpoint.

### **4.3 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 procedure 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.

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 only have a single PageServer which in case of failure will permanently crash the cluster. In the future we plan to substitute the single PageServer by a distributed solution with data replication so we can also tolerate multiple PageServer failures.

### **4.4 Network Partitioning**

The network might be temporarily partitioned 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. This is an open research question that will be studied in future work.

## 4.6 Resetting Nodes

As all OS context information is stored in the DSM and nodes can reboot in about 250ms in our test bed (see section two) we prefer to reboot the cluster when falling back to the last checkpoint. Enforcing a redraw of the desktop and of all its windows refreshes the screen.

One remaining issue is the restore of device states. Modern graphic cards for example store a plethora of data structures in their display adapter memory. After resetting a node this information is lost and any application depending on such external data must be able to restore its required data.

## 5 Conclusions and Future Work

Our running prototype shows that it is possible to store the full OS state within a DSM, even the OS code and data. This strategy simplifies global resource management and checkpointing. - but only if a sophisticated memory management and a strong efficient memory consistency model are provided. The latter is achieved by transactional consistency used for the first time in Plurix.

Resetting a cluster in case of an error to the last checkpoint is typically not a trivial task. By storing everything in the DSM, using restartable transactions, and being able to restart nodes in 250ms this task is not as difficult in Plurix as in traditional cluster systems. Smart buffers make device access within restartable transactions safe but unexpected cluster resets may still require that the application is able to restore device states. For example in case of a fall back to the last checkpoint it is unclear what textures must be reloaded to the graphics adapter. This topic is currently examined.

In future we plan to improve our global coordinated checkpointing strategy by writing asynchronously to disk. A replicated and distributed PageServer will be developed to improve write performance and to tolerate PageServer failures. Dependency tracking is also planned to avoid mandatory resetting of the cluster in case of a single node failure.

## References

- [Am94] C. Amza, A.L. Cox, S. Drwarkadas, and P. Keleher, „TreadMarks: Shared Memory Computing on Networks of Workstations“, in: Proceedings of the Winter 94 Usenix Conference, 1994.
- [Bin02] T. Bindhammer, R. Göckelmann, O. Marquardt, M. Schöttner, M. Wende, and P. Schulthess, “Device Programming in a Transactional DSM Operating System”, in: Proceedings of the Asia-Pacific Computer Systems Architecture Conference, Melbourne, Australia, 2002.
- [Cab95] A-M. Kermarrec, G. Cabillic, A. Gefflaut, C. Morin, and I. Puaud, “A Recoverable Distributed Shared Memory Integrating Coherence and Recoverability”, in: International Symposium on Fault-Tolerant Computing (FTCS-25), pages 289-298, June 1995.

- [Co96] J. Cordsen, M. Cordsen, A. Gerischer, B. Oestmann, F. W. Schröder, "Evaluation of the SODA DVSM Consistency Framework", Final deliverable, Esprit project SODA (9124), 1996, also in The SODA Project, Studien der GMD (nr. 301), ISBN 3-88457-301-2, Oct. 1996.
- [El99] 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.
- [Fre02] S. Frenz, "Persistenz eines transaktionsbasierten verteilten Speichers", diploma thesis at the University of Ulm, Germany, 2002.
- [Goe02] R. Goeckelmann, M. Schoettner, M. Wende, T. Bindhammer, and P. Schulthess, "Bootstrapping and Startup of an object-oriented Operating System", in: Proceedings of the European Conference on Object-Oriented Programming, Malaga, Spain, 2002.
- [Goe03] R. Goeckelmann, S. Frenz, M. Schoettner, P. Schulthess, "Compiler Support for Reference Tracking in a type-safe DSM", in: Proceedings of the Joint Modular Languages Conference, Klagenfurt, Austria, 2003.
- [Kan01] S. Kannan, M. Roberts, P. Mayes, D. Brelsford, and J. F. Skovira, "Workload Management with LoadLeveler", IBM RedBook, ISBN 0738422096, 1st ed., 2001.
- [Kerr] <http://www.kerrighed.org>
- [Li88] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", in: Proceedings of the International Conference on Parallel Computing, 1988.
- [MP97] 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.
- [Mos93] D. Mosberger, "Memory Consistency Models", Technical Report TR93/11, Department of Computer Science, University of Arizona, 1993.
- [Mos] <http://www.mosix.org>
- [Sch02] M. Schoettner, "Persistente Typen und Laufzeitstrukturen in einem Betriebssystem mit verteiltem virtuellem Speicher", PhD thesis, Ulm University, Germany, 2002.
- [We02] M. Wende, M. Schoettner, R. Goeckelmann, T. Bindhammer, P. Schulthess, "Optimistic Synchronization and Transactional Consistency", in: Proc. of the IEEE International Symposium on Cluster Computing and the Grid, Berlin, Germany, 2002.
- [WF90] Kun-Lung Wu and W. Kent Fuchs, "Recoverable Distributed Shared Virtual Memory", in: IEEE Transactions on Computers, 39(4):460-469, April 1990.
- [WG92] N. Wirth and J. Gutknecht: "Project Oberon", ACM Press, Addison-Wesley New York 1992.