

Transactional Cluster Computing

Stefan Frenz, Michael Schoettner, Ralph Goeckelmann, Peter Schulthess
frenz@vs.informatik.uni-ulm.de

Abstract

A lot of sophisticated techniques and platforms have been proposed to build distributed object systems. Remote method invocation and explicit message passing on top of traditional operating systems are complex and difficult to program. As an alternative the distributed memory idea simplifies and unifies memory access, but performance drawbacks caused by expensive distributed locking mechanisms obviated this approach for cluster computing. To avoid these slowdowns more and more weak consistency models try to avoid these slowdowns by burdening the programmer with explicit synchronization. Our research group has developed a fast transaction system for a distributed heap that guarantees semantically correct access to shared memory.

Keywords: distribution, consistency, transactions, shared memory, operating system.

1 Introduction

Traditional distributed shared memory systems offer implicit replication giving rise to the question about consistency. Many strategies have been proposed to deal with the trade-off between strict consistency models, which enforce transparent replication, and relaxed models, which try to improve performance by accepting reads to outdated data and burdening concurrency control to the application. Relaxed consistency with its contingently appearance of inconsistent data values is unacceptable for sharing pointers or heap structures. On the other hand, guarding each access to shared memory lessens the benefit of cluster computing because of its immense overhead and latency.

This paper describes an alternative approach to a cluster network protocol, that supports strong consistency with implicit communication for a distributed heap or object storage with equated cluster participants. The symmetric cluster design enables fully concurrent execution without a dedicated coordinating node and the transactional consistency model provides a natural application behavior without explicit communication or synchronization. Semantical groups are executed as in atomic transactions, and the speculative memory access successfully hides the latencies normally inherent to network. High performance for parallelized algorithms is combined with reliable values allowing even pointer and heap management without explicit synchronization by the programmer. We developed a standalone operating system for a lean testing environment, which is described in section 2.1 together with the semantics of optimistic transactions in section 2.2. The design and implementation algorithms of a protocol for transaction support in a equated cluster environment are presented in section 3.1, which is followed by evaluation and comparison in section 4.

2 Testing Environment

2.1 Plurix

Plurix was created at the University of Ulm as standalone operating system for research and educational purposes mainly considering distributed algorithms like matrix operations or ray tracing and distributed applications like cooperative working. Main aspects are the development of protocols and the investigation of programming concepts for distributed systems. The central event loop and the cooperative multi-tasking are adopted from the innovative Oberon system developed by Wirth and Gutknecht [Wir92], combined with comfortable data sharing done with distributed shared memory. Using the Java language with some extensions for operating system's requirements offers type-safe and easy programming and access to a wealth of already written software. Plurix uses a specially developed compiler [Sch02], which compiles Java sources directly into Intel machine code and abandons the Java Virtual Machine.

Currently, Plurix communicates via a Fast Ethernet in a local area network, but is ready to switch to Gigabit Ethernet. The distributed shared memory uses Transactional Consistency [Wen03] to enable a full featured heap avoiding dangling references. The concept is proven to correctly execute parallel transactions [Wen02]. The current implementation is optimized in respect of page-sized granularity to enable utilization of hardware accelerated memory management of current x86 computers. However, the protocol design is not bound to this approach.

2.2 Speculative Transactions

As a cooperative multitasking operating system Plurix avoids the overhead typically encountered in preemptive environments. Programs running in the Plurix environment are divided into small execution blocks, which are executed atomically and usually integrate few semantically complete units. Each atomically executed block is called transaction and conforms to the well known ACID paradigm [Hae83], [Dad96]:

- Each transaction either succeeds or is completely rolled back (atomicity), which is guaranteed by the memory management [Goe03]. One reason for a roll-back may be a self-abort of the transaction, for example when the transaction has detected an error condition and wishes to leave the heap unchanged. The second reason for a roll back is a forced abort in case of an access collision between two transactions.
- A transaction starts with consistent state of memory and after committing leaves it in a consistent state (consistency). All data and code [Goe03] resides in the distributed heap, and the heap may be changed only from within a transaction. This is provided by the kernel, which manages the heap with only few local state variables, which can not be accessed by a user transaction.
- The results of a transaction are not propagated until its commit (isolation), which means: before a successful commit the changes done by a transaction are only visible to this transaction. In case of an abort this isolation prevents cascading aborts and inconsistency of memory. The programmer can define semantic blocks, which require atomic calculation or should not interfere with others [Fre04].

- The results of a transaction survive the transaction even in case of a subsequent system error (durability). In databases, durability is ensured by saving each transaction's results to disc, but for cluster operations the term “durability” is more flexible and may be specified for each environment and application [Fre02]. The realization of durability was tested in two ways. First, we replicated within the cluster. Then, we made the same test with a pageserver that holds and buffers committed pages. Both cases provided an adequate measure of durability, but each proved to be advantageous for different applications.

At the end of each transaction the addresses of pages modified by this transaction are published in a semantically atomic way to avoid interleaving between multiple ending transactions. The usual way is just to send the addresses, which results in an invalidate-semantic: outdated data is simply removed and will be requested on demand. Another strategy is to send the data of each modified page along the address, which corresponds to update-semantic: outdated data is replaced by fresh data and immediately accessible. In both cases every concurrently running transaction must check the received writeset against its own read- and writeset to detect possible memory access conflicts. A collision will abort the local transaction, which does not require any network communication because of the isolation property of transactions.

3 Transactional Consistency Protocol

3.1 Design

The protocol must provide at least basic exchange and control mechanisms for communication between nodes. For transactional consistency this includes but is not limited to management of cluster-time, transfer of page data, reliable and atomic delivery of writesets and algorithms for error detection and recovery.

As far as possible the protocol should handle requests without additional state machines and should answer all requests directly in the interrupt service routine. This leads on the one hand to a efficient and fast response behavior and on the other hand to less indirections in the code and therewith better readability. The complexity seems to grow for the protocol caller if the protocol does not handle timeout errors itself, but as the caller can not proceed without completely fulfilled requests, there has to be a wait anyway, which can be easily equipped with timeout functionality.

LAN communication hardware typically offers non-reliable multicasts and unicasts. Because of packet loss or packet overtaking inside switches and network cards, the protocol must validate each packet. Therefore each packet contains a time-stamp from the sending machine, which lets the receiving machine decide whether the packet is valid, is outdated, or indicates a critical packet miss. The transition from one time-slot to the next is implicitly done with the first and last packet of a commit, because the writeset, which is delivered inside the commit-packets, is critical to the system consistency and has to be processed atomically. During the commit phase no other packet type is allowed, as data is in a transient state.

3.2 Algorithms for Implementation

The packet types required for consistent communication will be discussed in this section. Each packet contains information about the current logical time of the sending node, the membership to a cluster and a packet type. Packets may contain additional data of fixed or variable length, but in order to remain compatible with Fast Ethernet, the overall size may not exceed 1536 bytes.

General packet layout

Fast Ethernet is able to transmit specially tailored packet frames, so there would be no need of IP headers or similar. But in order to communicate across routers and to coexist with other nodes in a heterogeneous network, all packets contain an IP header. The Fast Ethernet header consists of destination and source MAC address (each 6 bytes) and a frame type (2 bytes) which is fixed to IP. The standard IP-header with 20 bytes includes source and destination IP and the protocol type, which is fixed to an otherwise unused value. As mentioned above, each packet carries a time stamp from the sender (commit number, 8 bytes) and an identifier for the particular heap or cluster, which is in our implementation a 8-byte-value. Furthermore there is a type with subtype (2 bytes) and a parameter for this packet type (4 bytes), which results in a header of 22 bytes. All headers together are 54 bytes. Depending on the packet type there may be additional data bytes up to 1444 bytes yielding a maximum packet size of 1498 bytes.

Validation of a received packet

Each packet contains the logical time of the sender, which is compared to the local logical time with several results possible: If the received time stamp matches the local time stamp, the packet is valid. Otherwise the packet is invalid, and the higher stamp indicates the current time. If the received time stamp is lower, the sender may have missed a writeset and must be notified. A received time stamp greater than the local time indicates that the local machine has lost a writeset, which must then be requested explicitly. Figure 1 at the end of this chapter shows the corresponding flowchart.

Alive Request and Alive Acknowledge

To determine the presence of other nodes (e.g. at start-up), a packet with type “alive request” is sent as multicast. Each running node in the cluster with matching heap number will answer this request with “alive acknowledge”. A starting node has no information about the current time in the cluster, so the sender-time is cleared. As the acknowledge packet contains the current time, the requesting node can update its time.

Page Request and Page Data

Page-based distributed shared memory as implemented in Plurix typically uses paging across the network. During execution a transaction may access a not present page at any time resulting in a page fault. The page fault handler verifies the address of the accessed page and requests the page from the cluster by sending a packet “page request” with the corresponding page address. For object based distributed memory systems the page number is replaced by an object id correspondingly.

The node owning this page will answer the page request with one or more packets of type “page data”, which all contain a CRC for the complete data to ensure correct transmission and reassembly of the transferred page. As the page size is 4096 bytes and the maximum packet size of Fast Ethernet frames is 1536 bytes, the data can not be sent forthright in a single frame and would require three packets without additional algorithms. Because statistics showed that many pages are completely cleared, there is a special message “page empty”. Checking of page emptiness is done during calculation of the CRC. In object-based systems the variable size of packets has to be considered similarly.

Since the transmission of a large number of packets consumes the bandwidth of the network, optional compression should be supported. Even though compression may spare bandwidth, it consumes CPU-time on the sending and receiving host and should only be used with sufficient advantage. In our implementation the page to send is compressed in temporary memory and the size of the compressed block is checked. If compression does not save at least one packet, the page is sent uncompressed even if compression is enabled because the network bandwidth saved is not significant.

Instead of retransmitting a single frame in case of uncommon packet loss, the requesting node simply re-requests the page after a timeout, so there is no additional bookkeeping or buffering inside the answering node.

Token Request and Token Granted

A terminating transaction attempts to commit by sending its writeset to all participating nodes. Since a transaction may conflict with another, one commit operation has to complete before the next commit can be started. To guarantee this Plurix implements a token mechanism using the packet type “token request” and the corresponding answer “token granted”.

To avoid token loss in case of a packet loss, a node sending the token stores the recipient of the token in a private local memory location, so the token can be resent to this node if this node requests the token again.

Writeset Frame and Recovery with Writeset Request

The writeset which is broadcast at the end of a transaction contains the addresses of all modified pages. The committing node becomes owner of all modified pages and all outdated copies on other nodes must be discarded. Because of the limitations of the maximum packet length, only up to 370 addresses are sent in a single Fast Ethernet packet. If the transaction modified more than 370 pages, the writeset must be split into multiple frames. Although there may be several packets, the writeset must be evaluated atomically to ensure consistency of memory.

If a writeset frame is lost by a node or the network, the affected node has to request the writeset again. Therefore each node must store previously sent writesets for a short time, until most likely all other nodes have received them.

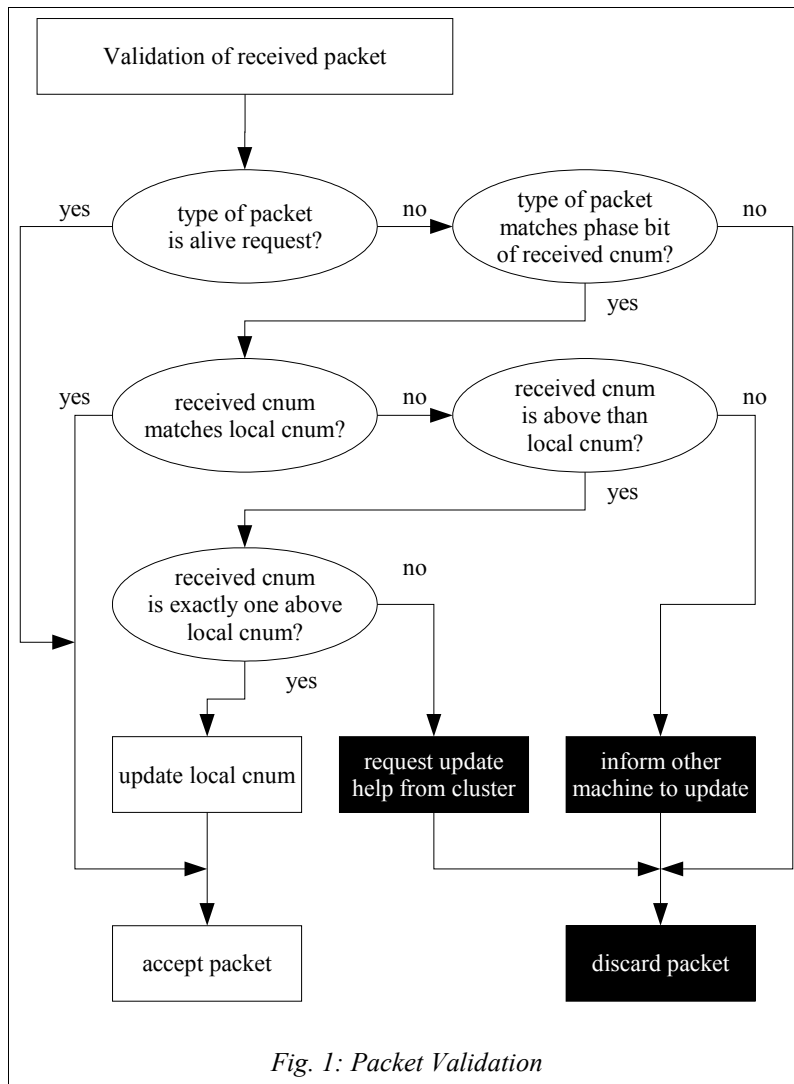


Fig. 1: Packet Validation

3.3 Fairness

The basic implementation does not support fairness for repeatedly aborted transactions. But the forward validation scheme allows sophisticated commit election strategies instead of simple “first wins” strategy. A protocol for improved fairness will need to take into account detailed information about the conflicting transactions to choose the optimal set of surviving transactions. There are many decision strategies about which set of transactions should survive and which set should be aborted. The metrics consider completely different application's behavior, which comprise different amount of parameters. Although the fairness of the decision generally increases by adding more parameters, it depends on the type of running applications which algorithm performs best.

3.4 Checkpointing

Transactional Consistency offers a natural opportunity to checkpoint a cluster: The system architecture is prepared to discard modifications and to request current versions from the cluster, and the isolation property of transactions will let the cluster computation proceed during collection of data for a checkpoint. Of course only the distributable part of memory is checkpointed by the cluster, so each node has to make sure that local state can be restored from the distributed memory or can be regenerated by the kernel or by the application. To save also the local state in the distributed memory is neither trivial, because kernel and device states usually are not distributed, nor fast enough, because the local kernel and device drivers change their state frequently and quickly, which would make the cluster coordination to a bottleneck. Fortunately many of these states are not critical to the application and can be generated from small portions of basic information.

For example much information about an open file is kept in the kernel, but really critical to the application is only the filename, the position inside the opened file and the access rights. Usually the application only gets a handle to the file and all other information is kept in the kernel memory, which would require special attention during checkpointing. By moving the required information to the shared heap enables the application to access an opened file even after a fallback, because the kernel is able to restore the information from filename, position and access rights.

4 Evaluation and Comparison

All tests are done with a cluster of twelve single CPU machines with Fast Ethernet connection over an Allied Telesyn AT-8024 switch. The nodes are Athlon XP2500+ with 512 MB DDR-RAM equipped with 3Com 905 card on an Asus A7V8X-X.

4.1 Network Measurements

The usable bandwidth is important to cluster systems and depends on the operating systems architecture, the protocol stack and the network interface. Therefore it is at first interesting to know the overhead for transferred data and on the other hand to know the maximum achievable throughput for page data transfer. The page data transfer packets are in the current implementation split up into three packets with 1400, 1400 and 1296 bytes of payload, if the page is not empty. Otherwise a single frame with no (extra) payload is enough. Additionally for each packet a header of 54 bytes is required, containing the information for the Ethernet, IP and the DSM. The Plurix-Protocol contains 22 bytes and is treated as payload because of the information about current time and packet type (like “requested page is empty”). The behavior of our implementation shows that Fast Ethernet works well for packets with more than 300 bytes and allows a bandwidth above 11 megabytes per second. The theoretical maximum transfer rate of Fast Ethernet with 100 megabits per second is 12500 kilobytes per second, which is assumed to be 100% for utilization in the following table. As for all packets there is an additional preamble and CRC on the Ethernet medium, the overhead increases with smaller packets.

<i>Ethernet</i>			<i>Payload</i>		
<i>Packetsize bytes/packet</i>	<i>Throughput kB/sec</i>	<i>Utilization of Ethernet</i>	<i>Data bytes/packet</i>	<i>Throughput kB/sec</i>	<i>Utilization of Packet</i>
1500	11970	95,76%	1468	11714	97,87%
1000	11870	94,96%	968	11490	96,80%
576	11663	93,30%	544	11015	94,44%
272	11145	89,16%	240	9834	88,24%
160	6291	50,33%	128	5033	80,00%
96	3513	28,10%	64	2342	66,66%
64	2381	19,05%	32	1191	50,00%

Data pages are transferred at about 11.7 MB/s, which means about 2900 pages with data can be transmitted per second on the current infrastructure. Empty pages are transferred at about 2.4 MB/s resulting in a maximum of about 37200 empty pages per second. Network switches aggregate the bandwidth, so the measured values are the limit of direct one-to-one connections of which many can take place at the same time.

4.2 Protocol and Transaction Measurements

A critical time for a transactional system is the overhead at the beginning and at the end (both aborting and committing) of a transaction. As we have implemented a first wins strategy with a token, the token latency becomes an interesting value, too. For a page-based system, the minimum time between page request and complete reception of a page is important as well as the maximum throughput (see section above).

	<i>minimum</i>	<i>normal</i>	<i>upper</i>
<i>begin of transaction</i>	15 μ s	17 μ s	25 μ s
<i>end of transaction, abort / commit</i>	6 / 16 μ s	12 / 18 μ s	24 / 26 μ s
<i>latency for empty / filled page</i>	97 / 780 μ s	99 / 783 μ s	102 / 797 μ s
<i>token latency</i>	66 μ s	71 μ s	94 μ s

The begin of transaction is independent from the cluster and has an overhead depending on the previous transaction, because all page table entries have to be reset and initialized. Most transactions are short and have only few page table entries to be modified, so this usually takes about 16 microseconds but may take up to 25 microseconds after a a data intensive transaction.

The end of a transaction may be either an abort with resetting all information or a commit with publication of all modifications. The measured times give both values if the token is already at the committing node. Otherwise there is an additional call to get the token, which is around 70 microseconds.

As mentioned the page latency is very important to page based distributed systems. Each access to a not present page requires a page request with appropriate answer from the cluster. In the measurements the time taken to request and answer one page is given for both a filled and an empty page. The filled page needs three packets with 4258 bytes ($4096+3*54$), the empty page one packet with 54 bytes (see section 4.1).

To verify these values we made an additional test with 1000 empty transactions committing on a single node that already possesses the token. The time needed was 32110 ms, straight below the normal value of $n*(BOT+EOT)$ coming to 31-35 ms.

4.3 Application Measurements

Currently a raytracer and the well-known successive-over-relaxation algorithm were implemented to test the programming model and the behavior of our system. The raytracer as fully parallelized application performs very well with nearly linear speed-up. With larger image sizes the speed-up grows, as the initial distribution of the scene and matrix is less relevant compared to the computation.

<i>Raytracer Execution Time in Milliseconds</i>						
<i>Nodes</i>	<i>1024x768</i>	<i>1448x1086</i>	<i>2048x1536</i>	<i>2896x2172</i>	<i>4096x3072</i>	<i>5792x4344</i>
1	78067	156004	312121	623460	1247697	2494528
2	39780	79136	157716	314178	625691	1249590
4	19988	39624	79088	157293	314168	626804
8	10231	19961	39842	78883	157073	313449
12	7381	13910	27448	53328	105430	210028

<i>Raytracer Speedup</i>						
<i>Nodes</i>	<i>1024x768</i>	<i>1448x1086</i>	<i>2048x1536</i>	<i>2896x2172</i>	<i>4096x3072</i>	<i>5792x4344</i>
1	1,000	1,000	1,000	1,000	1,000	1,000
2	1,962	1,971	1,979	1,984	1,994	1,996
4	3,906	3,937	3,947	3,964	3,971	3,980
8	7,630	7,815	7,834	7,904	7,943	7,958
12	10,577	11,215	11,371	11,691	11,834	11,877

The successive-over-relaxation needs much more communications in comparison to the raytracer and has a single-calculator-multi-wait behavior, so linear speed-up is not reached. To provide comparability to other systems, we made measurements under Linux with JPVM [jpvm] and Aleph [Her99], [aleph] in the same cluster, too. Both other systems use the Sun Java Environment. The following tables show these results, where for each matrix size the relative speed up to the single node time and to the Plurix single node time is given to provide both internal speed-up and comparison.

<i>SOR time</i>	<i>2048x2048</i>			<i>4096x4096</i>		
<i>Nodes</i>	<i>Plurix</i>	<i>Aleph</i>	<i>JPVM</i>	<i>Plurix</i>	<i>Aleph</i>	<i>JPVM</i>
1	121,48	533,02	676,78	458,49	2122,37	2721,14
2	70,10	281,06	367,39	250,70	1084,08	1414,04
4	45,22	155,92	191,27	142,57	566,66	737,63
8	36,66	155,66	118,65	98,31	330,80	413,52
12	37,52	120,52	100,20	87,35	270,01	326,92

<i>SOR sdup</i>	<i>2048x2048</i>			<i>4096x4096</i>		
<i>Nodes</i>	<i>Plurix</i>	<i>Aleph</i>	<i>JPVM</i>	<i>Plurix</i>	<i>Aleph</i>	<i>JPVM</i>
1	1,00	0,23/1,00	0,18/1,00	1,00	0,22/1,00	0,17/1,00
2	1,73	0,43/1,90	0,33/1,84	1,83	0,42/1,96	0,32/1,92
4	2,69	0,78/3,41	0,64/3,54	3,22	0,81/3,75	0,62/3,69
8	3,31	0,78/3,42	1,02/5,70	4,66	1,39/6,42	1,11/6,58
12	3,24	1,01/4,42	1,21/6,75	5,25	1,70/7,86	1,40/8,32

The JPVM shows very good internal speed up as Aleph does, but the overall performance is about five times slower than Plurix. Therefore the speed-up can be better easily, because the time to communicate is much less than the time to calculate. This results in a faster execution on two nodes running Plurix than twelve nodes running JPVM. For Aleph and JPVM the communication is done explicitly, whereas the programmer does not have to manage the data exchange in Plurix. For the small matrices the Plurix system has the maximum speed-up at 8 nodes, whereas the maximum speed-up for the bigger matrix is not reached with even twelve nodes. Both JPVM and Aleph never show this point of return at those measurements, but as the overall execution time is much higher than with Plurix, the communication is not expected to be the bottleneck.

5 Related Work

Transactions are well known in the database world for more than two decades and are implemented for example in PostgreSQL [Sto87], MySQL [mysql] and Oracle [oracle]. The transaction paradigm was overwhelmingly successful in this context, and A.P. Black [Bla90] advocated its transfer to operating systems. Plurix supports transactions both for single station and cluster set-ups.

The idea of distributed shared memory systems was presented by L. Keedy in 1985 [Kee85]. There are several page-based systems with distributed shared memory like IVY [Li88], Mirage [Fle89] and TreadMarks [Kel94] with different consistency models. Mungi [Hei94] uses 64 bit processors for a distributed single address space secured with password capabilities [Hei98], the consistency is guaranteed with single-

owner and write-invalidate semantic. As an alternative Plurix integrates semantically grouped accesses into a transaction and validates them in a forward validation scheme with optimistic synchronization.

Systems like Eiffel** [Hug93], Thor [Lis99] or PerDIS [Fer99] provide transactions with distribution, but do not integrate the transaction concept into the operating system and partially use a server/client architecture. In contrast Plurix provides a flat distribution model on the kernel level.

6 Conclusion and Future Research

Transactional Consistency helps programmers to write distributed applications by providing implicit and automatic communication or synchronization. The memory and transaction management in the kernel guarantees consistent and race-free parallel execution of applications with only small overhead. The implementation of the protocol is lean and efficient, it lends itself to high performance cluster computing with low latency and near maximum throughput as shown in the measurements. Future work will be done on optimizing the protocol for Gigabit Ethernet and to include different fairness strategies.

7 References

- [aleph] <http://www.cs.brown.edu/people/mph/aleph/>
- [Bla90] A.P. Black: "Understanding Transactions in the Operating System Context". Proc. of the 4th workshop on ACM SIGOPS, p1-4, 1990.
- [Dad96] P. Dadam: "Verteilte Datenbanken und Client/Server-Systeme". Springer-Verlag, Heidelberg, 1996.
- [Fer99] P. Ferreira et al.: "PerDiS: design, implementation, and use of a PERsistent DIstributed Store". INRIA RR 3525, Rapport de Recherche, Institut National de Recherche en Informatique et Automatique, Rocquencourt, France und Lecture Notes In Computer Science, vol 1752, p427-452, 1999
- [Fle89] B.D. Fleisch et al.: "Mirage: A Coherent Distributed Shared Memory Design". Proc. of 14th ACM Symp. on Operating Systems Principles, 1989.
- [Fre02] S. Frenz: "Persistenz eines transaktionsbasierten verteilten Speichers". Diploma-Thesis at the University of Ulm, Distributed Systems, 2002.
- [Fre04] S. Frenz, M. Schoettner, R. Goeckelmann, P. Schulthess: "Performance Evaluation of Transactional DSM". Proc. of the 4th IEEE/ACM Intern. Symposium on Cluster Computing and the Grid, Chicago, 2004.
- [Goe03] R. Goeckelmann, M. Schoettner, S. Frenz, P. Schulthess: "A Kernel Running in a DSM - Design Aspects of a Distributed Operating System". Proc. of the IEEE Intern. Conf. on Cluster Computing, Hong Kong, 2003.
- [Goe04] R. Goeckelmann, M. Schoettner, S. Frenz, P. Schulthess: "Plurix, a Distributed Operating System extending the Single System Image Concept".

Proc. of the IEEE Canadian Conference on Electrical and Computer Engineering, Niagara Falls, Canada, 2004.

- [Hae83] T. Haerder, A. Reuter: „Principles of Transaction-Oriented Database Recovery“, Computing Surveys 15, 4, p287-317, December 1983.
- [Hei94] G. Heiser, K. Elphinstone, S. Russell, J. Vochtelo: “Mungi: A Distributed Single Address-Space Operating System”. Proc. of the 17th Annual Computer Science Conference, ACSC-17, 1994.
- [Hei98] G. Heiser et al.: “The Mungi single-address-space operating system”. Software: Practice and Experience, vol 28, p901-928, August 1998.
- [Her99] Maurice Herlihy: “The Aleph Toolkit: Support for Scalable Distributed Shared Objects”. Proc. of the 3rd Intern. Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications, 1999.
- [Hug93] C. McHugh, V. Cahill: “Eiffel**: An Implementation of Eiffel on Amadeus, a Persistent, Distributed Applications Support Environment”. TOOLS Europe '93 Conference Proceedings, p47-62, 1993.
- [jpv] <http://www.cs.virginia.edu/~ajf2j/jpvm.html>
- [Kee85] J.L. Keedy, D.A. Abramson: “Implementing a large virtual memory in a Distributed Computing System”. Proc. of the 18th Annual Hawaii International Conference on System Sciences, 1985.
- [Kel94] P. Keleher, A.L. Cox, S. Dwarkadas, W. Zwaenepoel: “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems”. Proc. of the Winter 1994 USENIX Conference, 1994.
- [Li88] K. Li: “IVY: A Shared Virtual Memory System for Parallel Computing”. In Proceedings of International Conference on Parallel Processing, 1988.
- [Lis99] B. Liskov, M. Castro: “Providing Persistent Objects in Distributed Systems”. Proc. of ECOOP'99, 1999.
- [mysql] <http://www.mysql.com/>
- [oracle] <http://www.oracle.com/technology/documentation/index.html>
- [Sch02] M. Schoettner: “Persistente Typen und Laufzeitstrukturen in einem Betriebssystem mit verteiltem virtuellen Speicher”. PhD-Thesis at the University of Ulm, Distributed Systems, 2002.
- [Sto87] M. Stonebraker: “The Design of the POSTGRES Storage System”. Proc. of the 1987 VLDB Conference, Brighton, September 1987.
- [Wen02] M. Wende et al.: ”Optimistic Synchronization and Transactional Consistency”. Proc. of the 4th Intern. Workshop on Software Distributed Shared Memory, Berlin, 2002.
- [Wen03] M. Wende: “Kommunikationsmodell eines verteilten virtuellen Speichers”. PhD-Thesis at the University of Ulm, Distributed Systems, 2003.
- [Wir92] N. Wirth, J. Gutknecht: “Project Oberon”. ACM Press, New-York, 1992.