Foreword

This report is the result of a master thesis performed by Steffen Holthe and Jan Steinar Kvilesjø in the 5th year of a Masters Degree at the Norwegian University of Science and Technology (NTNU) in Trondheim, Department of Computer and Information Science, 2005. The thesis' title is "Highly available database clusters – Repair with large segments". The thesis will give an answer to the following problem:

In existing highly available database clusters, node repair is an expensive operation due to use of the TCP/UDP/IP protocols. The candidates shall create a prototype of a DBMS, using InfiniBand and RDMA for node repair. InfiniBand and RDMA open for the possibility to transfer large segments over a high speed network and in this way improve the performance. The model will simulate such a system to see how use of this technology affects the performance and availability compared to existing solutions. Based on this solution, the candidates will try to discover what segment size that gives the best overall performance for the system.

The report is written for people with a general database background and interest in using new network technology in highly available database clusters. It is performed on behalf of Sun Microsystems and Professor Svein Erik Bratsberg. We would like to thank Professor Bratsberg and Mr. Rune Humborstad from Sun Microsystems for good help and counseling during our work. We would also give a special thanks to Linda Ring for help with spelling and grammar and Dr. Terje Brasethvik for help with structuring and quality assurance of the thesis.

Trondheim, Friday 3rd of June 2005

Steffen Holthe	Jan Steinar Kvilesjø

Abstract

The goal for this master thesis is finding trends and behaviors in a highly available database cluster using InfiniBand and RDMA. This will be used to optimize configuration of the segment size in such systems. To find these trends and behaviors, a mockup model has been developed. The model consists of a simple DBMS that uses only the main memory for storing data and a checkpoint method to repair nodes after a node failure. During a repair, the model simulates use of InfiniBand and RDMA during checkpointing.

To simulate clients connecting to and using the database, the model includes write operations on the database and measures how many write operations it can process per second. During a repair in a database cluster, one node will flush all of its data to a new node. This is done in small batches, just like in a checkpoint. In this model, it is simulated by a checkpoint module continuously simulating flushing data from one node to another. When the checkpoint is flushing a small part of the database fragment, the model uses Copy on Write to prevent lockout for the transactions.

When a node fails the system is repaired by a second node which takes over as fast as possible. The second node must process transactions while transferring all of its data to a spare node. To achieve fast repair time, the system should transfer as big segments as possible. The problem with big segments is that it takes a long time to perform Copy on Write on them.

With the mockup model, the repair time is measured, how many write operations are being performed per second and CPU usage depending on the segment size and number of clients using the database. This will give a good indication of what segment size is preferable.

The results from the thesis show that there are huge advantages by using state-of-the-art technology such as InfiniBand and RDMA for repair in highly available database clusters. This technology and optimal configuration improves the availability and this thesis gives an indication that the segment size should not be more than 1 % of the database size. With use of InfiniBand and RDMA using this configuration and physical repair, the availability reaches class 9.

Contents

Chapters

Graphical content

Graphical content without references are created by the authors.

Figures

Tables

Graphs

Code

Chapter 1

Introduction

This chapter gives a short explanation of the motivation for the assignment and an introduction to Suns HADB (High Available Data Base).

1.1 The assignment

The background for this assignment comes from Sun Microsystems and professor at NTNU, Svein Erik Bratsberg. They want to find out more about use of InfiniBand when checkpointing a database from one node to another, especially related to HADB. When severe errors occur or maintenance is performed in HADB, the whole database fragment is copied from one node to another to make the new node up to date (Left in Figure 1.1).

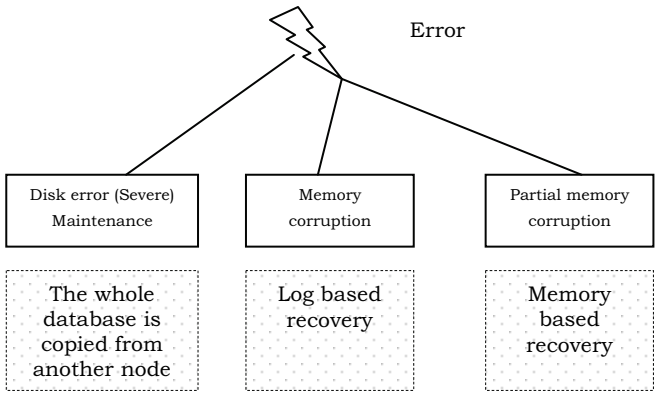


Figure 1.1: Error handling in HADB [HK04]

For this copy operation it is desirable to transfer as much data as possible at one connection because of the connection setup time which occurs for each connection when using RDMA over InfiniBand.

The transfer rate for InfiniBand in this thesis will be estimated at 8 Gb/s. When transferring a database from one node to another, one segment of the database is transferred at a time. Theoretically it would be preferable to have these segments as large as possible. The segment that is transferred will have to be marked in the database for Copy on Write inside the source node. Copy on Write is used to avoid blocking in the database which will lead to a significant fall in the throughput. Large database segments will create an expensive internal copy operation in the main memory and take a lot of resources. The main goal for this thesis is to collect enough data to be able to make a suggestion as to which segment size gives a high bandwidth for the copy operation between nodes and a high throughput inside the source node. The collected data will validate the hypotheses that will be presented in Chapter 4.

1.2 HADB

HADB was first developed at Telenor in the middle of the 1990s and was called Clustra [HK04]. Now the system is called HADB and is used in Sun Java Enterprise System which is a part of Suns highly available J2EE Application Server. The project had three main goals (a) Short response time – 15 ms for 95 % of the transactions, (b) Scalable throughput with an upper limit of at least 1000 TPS, (c) High availability, not more than 3 minutes downtime per year. The transactions in question were simple Telecom transactions.

HADB is built on a shared-nothing architecture which consists of several clusters with Sun UNIX, Windows 2000 or Linux computers. The different nodes are grouped into sites with communication through switches (Figure 1.2). Two phase locking is used for concurrency control. Tables are fragmented horizontally. There are primary nodes and hot-standby nodes. Hot-standby nodes are identical copies of the primary nodes. These nodes are able to take over if the primary nodes go down.

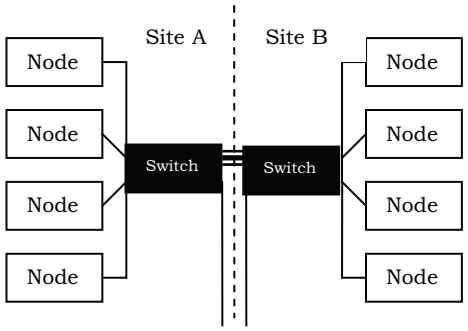


Figure 1.2: The HADB system [HK04]

During checkpointing HADB uses the following techniques; Fuzzy checkpointing, Steal and No-force. A fuzzy checkpoint is non-blocking. HADB uses two-level logging, one logical record log, and a node-internal physical log. It guaranties that the internal node log records are written to disk before the corresponding data pages. Steal means that it is possible to flush dirty page to disk before the transaction has committed. No-force means that even if the transaction commits this does not mean that dirty pages are forced to disk. In this way it is much easier to control the time and place for disk accesses. Redo during recovery starts on the penultimate checkpoint since no log records are written to the log after a checkpoint. Thus it is not possible to assure that the last checkpoint was performed successfully. This type of checkpoint is called penultimate fuzzy checkpoints.

The buffer manger, checkpoint manager and the disk interface cooperate to maintain writing and fetching of pages. The buffer manager uses the checkpoint manager to empty the dirty page buffer if necessary. The checkpoint manager scans this buffer for dirty pages at certain time intervals and pages found are written to disk. The disk interface receives a request to perform write or read operation. These operations are handled using asynchronous I/O. The page size used in HADB today can vary between 4 KB and 16 KB. The max record size is less than half of the block size.

1.2.1 The checkpoint module

The checkpoint mechanism (Figure 1.3) does not consider the database log and can be seen as a combined checkpoint and write-ahead mechanism. A special aspect with the buffer manager in HADB is that the checkpoint thread is the only thread that performs output of dirty pages to the disks.

This thread is activated at regular time intervals, either at certain times or when the buffer manager detects a certain amount of dirty pages.

Checkpoints are frequently written because this leads to a short recovery time after a possible crash. Dirty pages are flushed regularly to disk to avoid that the system will run out of non-dirty pages. This will create a delay for the system when dirty pages have to be written to disk to free up space.

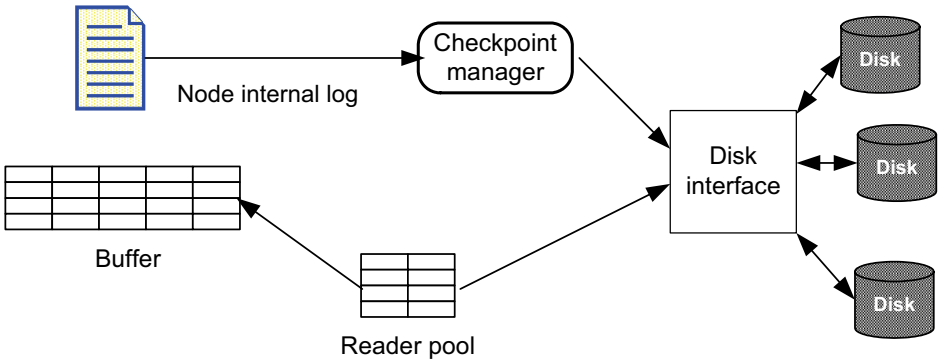


Figure 1.3: The checkpoint mechanism [HK04]

1.2.2 Repair with the checkpoint technique

HADB has several ways to handle errors. After a node failure the hot-standby node must copy all of its fragments over to a new node. An example of a node failure can be a disk error. This repair mechanism uses the checkpoint technique, but instead of writing to disk as it usually does, it transfers the data to the main memory on a spare node in the network. During this repair period the system must process transactions and transfer all the data to the spare node. This causes extra load on the node and increases the chances of double faults. Thus it is crucial that this repair time is as short as possible while providing access for the transactions. Repair can be performed in two ways, logically or physically.

1.2.3 Logical versus Physical repair

A clustered system can tolerate some level of faults. After a node crash and a following unsuccessful recovery the repair process reestablishes this fault tolerance level. Repair involves copying the data and log of one node to another unused node of the system. This is done by reading, transferring and inserting and can be done logically or physically.

The meaning of logical repair is that individual data records of the database are copied. The copy process reads these individual records from the database by a low-level cursor. Each record is copied into an outgoing message buffer. When the buffer is full, the message is transferred, typically containing a few records within one UDP (User Datagram Protocol) message.

Along with the data copy the log records are also copied. The new copy of the database is kept updated by applying each log record to the corresponding data record at the receiving node, given that the records have arrived. The individual data records and log records are transferred in parallel using rather small messages. In addition, it takes time to scan the database record by record and considerable time to insert a database record by record. Logical repair method is used in HADB today.

Physical repair is quite another process. In physical repair the database is copied in a rawer format and rather large segments may be copied as a whole. The advantage of this is that reading, transferring and inserting the database is much cheaper. If high speed networks are available, e.g. InfiniBand and RDMA, this may be very advantageous. However, when copying complete segments, consistency must be ensured by utilizing a physical log in addition to the logical log.

Logical repair is more flexible than physical repair concerning declustering strategy. Physical repair requires that source and target segments become equal. In logical repair two adjacent records may end at two different nodes, in case distributed sparing is used. Thus, the replication unit is individual records. However, in HADB mirrored declustering and separate spares are used. Another drawback of physical repair is that both data copies become physically equal and they thus may be more exposed to propagation of errors across nodes (E.g. software errors).

1.3 Availability

Availability is the fraction of time that a system performs requests correctly and within specified time constraints [GR93]. One system contains multiple modules and these modules can fail. Mean Time To Fail (MTTF) is referred to as the reliability of one node and is the average time from a module is set into service, until it fails. After a module fails, it has to be repaired. The time spent on this repair is measured as Mean Time To Repair (MTTR) and is referred to as maintainability. Availability for one module can directly be computed from the reliability and maintainability measures for the module. Availability *A* is given by:

$$A = \frac{MTTF}{(MTTF + MTTR)} \qquad (1.1)$$

System availability is usually expressed as a percentage. To apply this idea, suppose a system with an average of one unscheduled outage every 100 days, and the problem takes one day to fix. Such a high failure rate (1%) dominates the system availability. From the customer's perspective the system availability is 99 % (= 100 % - 1 %).

Gray and Siewiorek [GS91] groups systems into seven availability classes. If *A* is availability, the availability class *C* can be expressed as:

$$C = \left\lfloor \log_{10}\left(\frac{1}{1-A}\right) \right\rfloor \qquad (1.2)$$

In Table 1.1 the classes from 1 to 7 is listed. In 1980 a class 2 was considered well, in 1990 a class 3 and today a class 5 is considered well.

System type	Unavailability (min/year) U	Availability A	Class C
Unmanaged	52.560	90 %	1
Managed	5.256	99 %	2
Well-managed	526	99.9 %	3
Fault-tolerant	53	99.99 %	4
High-availability	5	99.999 %	5
Very-high availability	0.5	99.9999 %	6
Ultra-availability	0.05	99.99999 %	7

Table 1.1: Availability of typical systems classes. [GS91]

There can be various reasons for service unavailability. For the user of the service the reason is of lesser importance, whether it is software bug or an earthquake. Availability of service has a direct impact on the users business and the cost of having an unavailable service is an important factor.

This is why the availability issue is an important one. Systems like telephone, stock market systems and web shops have large costs and lost income during service unavailability. Even though precautions are taken, it is not possible to achieve 100 % continuous service availability. All methods for fault-tolerance only reduce the probability of failure and do not fully eliminate it.

1.3.1 The availability model

Since availability for highly available system will be close to 100 % it can be more convenient to use the unavailability figure. An unavailability of 10^{-6} is much easier to understand than an availability of 0.999999 (99.9999 %). Unavailability is calculated by one minus availability.

The total system unavailability is a sum of the unavailability caused by the various reasons. In a cluster the most common reason for failure is the node-node double-faults. To calculate availability for one module the Equation 1.1 is usually used. The availability for a complete cluster has a different equation. Torbjørnsen has developed a failure formula for complete clusters [Tor95]. To analyze the system behavior, he has developed a failure model which describes the failure state transition for node failures. The system for this model has N nodes and no dedicated spare node storing two replicas of all data and implementing self repair of lost replicas. The system is in state S_i, where $0 \leq i < N$, when at least one replica of all data still is available and fragment replicas lost by i node failures (not due to node failures) not have been reproduced by self repair. If both replicas of one or more fragments are lost due to node failures the system is in state S_a and is considered unavailable. N is the total number of nodes in the system.

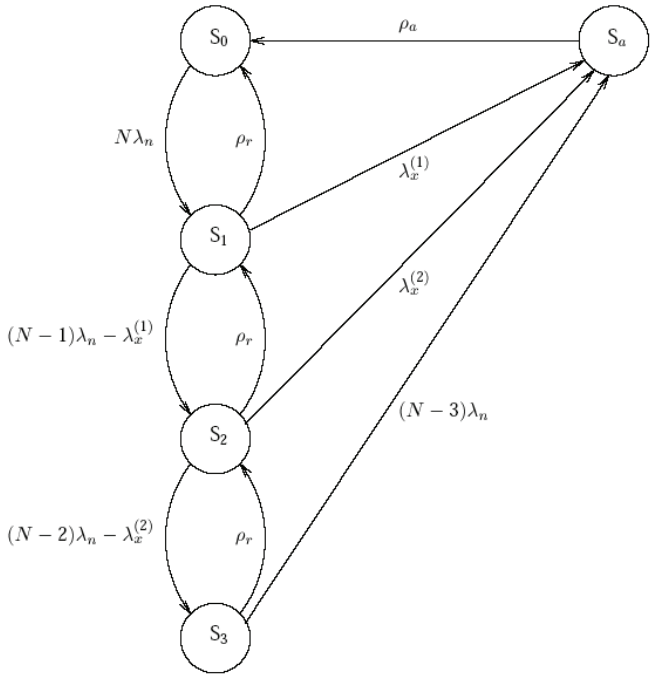


Figure 1.4: Failure state transition diagram assuming maximum three concurrent node failures. The intensity of transitions from state S3 to state Sa is given by the aggregate failure intensity of the remaining (N-1) nodes. [Tor95]

Figure 1.4 shows the transition diagram where a maximum of three failed nodes is allowed. The size and complexity of this model makes it difficult to understand the overall properties of the system. It is reasonable to believe that the system will spend most of its time in state S_0. The probability of being in any other states S_i, $i > 0$, will rapidly decrease with increasing i.

The system can under no circumstances provide full availability when the system is in the state S_a, since all replicas of some data have been lost. If the system does not provide on-line service during self repair the system is available only when in state S_0. This repair strategy will be called off-line self repair.

The opposite strategy, which is called on-line self repair, provides service during repair and results in only state S_a being an unavailable state. Unavailability is therefore the probability of any time being in a state when the system is unavailable.

The equation for on-line self repair is:

$$U_{NN,on-line} = \frac{\lambda_n^2 G Z_r N}{\rho_a \rho_r} \qquad (1.3)$$

λ_n = *Failure intensity for double failures in state* S_i
ρ_r = *Repair intensity*
ρ_a = *Recovery intensity*
G = *Failure intensity scaling factor for double failures*
Z_r = *Replica fanout*
N = *Number of nodes*

Since on-line self repair must use some of its resources processing user transactions, less is available to self repair. Thus, the repair is slowed down, reducing the repair intensity ρ_r, and therefore increasing unavailability.

Node failure, replica repair and system recovery are all exponentially distributed with parameters respectively λ_n, ρ_r and ρ_a. (Replica repair time will depend on the segment size and communication capacity that is used in the system). When a node has failed and not yet repaired, the failure rate for nodes sharing common fragments with it increases by a factor G to $G\lambda_n$. Z_r represents the replication strategy used by the system (replica fanout). When using a cluster with mirrored declustering this value is set to 1. One module in a mirrored declustering system is two nodes containing the same fragment of the database. Every primary node has an independent hot-standby node containing the exact same data fragments (Figure 1.5). When a node fails, it is repaired by transferring all the data from the primary or backup node to a new one. This can be done on-line, while the system is in use. During a repair the node has to copy all the data and process transactions from the users. Thus the load on one node will be extra large during repair and the probability for the node to cause double faults will increase. To minimize this fraction of time that the node is exposed with extra load, it is important that the repair is done as quickly as possible.

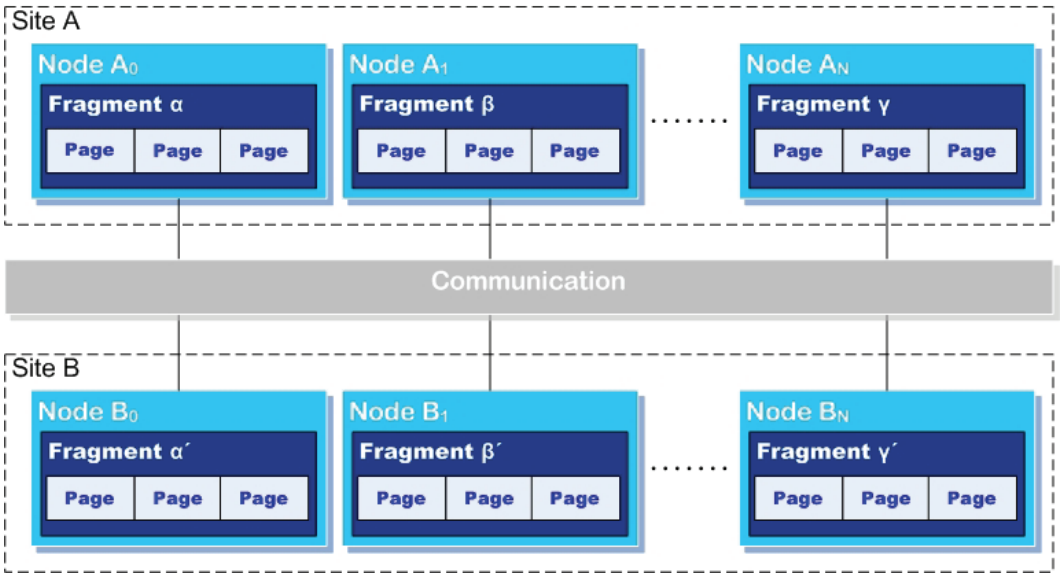


Figure 1.5: Cluster with mirrored declustering strategy

For a system with mirrored declustering the equation for unavailability is:

$$U_{NN,on-line} = \frac{\lambda_n^2 GN}{\rho_a \rho_r} \qquad (1.4)$$

This equation can calculate what unavailability the system will have with respect to the amount of nodes, probability of node faults and the time it takes to repair and recover the system. It will be used in Chapter 7.6 to show what kind of impact the segment size has on the availability of the system. The theoretical background will be presented in Chapter 2 and 3 to give an understanding of technology used in this thesis.

Chapter 2

Technological background

RDMA over InfiniBand is simulated in this master thesis. To get a good simulation, the technology must be understood. This chapter describes this technology, the Copy on Write and Java Buffer technology. The last two are used in the mockup model of the DBMS.

2.1 InfiniBand

InfiniBand is developed by the InfiniBand Trade Association which is lead by a steering committee composite of eight member companies. In this chapter the background and motivation for InfiniBand will be described before the technical explanation of the concept.

2.1.1 Background

InfiniBand came alive in 1999 as a suggestion to solve future communication problems related to the limited transaction rate throughout the PCI port [Kim04a]. In 1999 there were high speed communications through fiber optics, which was an expensive solution. InfiniBand is a high speed serial point-to-point link through copper. Instead of trying to drive lots of wires in a shared bus in parallel, a single wire at much higher speed is used. After a while this physical approach was coupled with Remote DMA (RDMA) and the Queue Pair (QP) model of asynchronous operation, borrowed from the Virtual Interface Architecture (VIA).

The InfiniBand Trade Association was created as a merge between the two organizations Next Generation I/O and Future I/O plus Microsoft (Today this association consists of Dell, Hewlett-Packard, IBM, Intel, Lane15 Software, Mellanox, Network Appliance and Sun Microsystems).

The goal for this organization was to connect all servers on the internet using InfiniBand. They claimed that they would replace PCI in I/O, Ethernet in the machine room, a cluster interconnect and fiber channel with one high capacity InfiniBand fabric and a single administration scheme.

Today the support for InfiniBand is evolving e.g. Linux, which supports InfiniBand with the release of kernel 2.6.11 [Sla05] and the specification of InfiniBand v.1.2 lays the groundwork to take InfiniBand up to 120 Gb/s [ITA04]. InfiniBand is also starting to make gains as a cluster interconnect. This is particularly true in latency sensitive applications (e.g. DBMS). In this market it has a latency edge over Ethernet.

2.1.2 Theoretical explanation of the concept

The links in InfiniBand are point-to-point (switched), bi-directional using 2.5 Gb signaling rate and 8b/10b encoding [Kim04b]. The basic 1x cable is capable of 250 MB/s in both directions simultaneously. The other bundle sizes are 4x (1 GB/s), 12x (3 GB/s). Release 1.2 adds an 8x bundle and allow for the signaling speed to be doubled (DDR) or quadrupled (QDR). The links can be implemented in a printed circuit board (20+ inches), copper (10 meters) and fiber (up to 10 kilometers). InfiniBand host adapters, called Host Channel Adapters (HCA), have a protocol processing engine in them that implements a hardware queue to accept commands for each communication endpoint in the adapter. Further, there are other queues which notify the adapter user when commands are done. Queues are used so the commands can be performed asynchronously by the hardware without a need to wait for the operations to complete. The commands send and receive messages or perform RDMA. A network using InfiniBand consists of three components (Figure 2.1), HCA, TCA (Target Channel Adapters) and a InfiniBand switch.

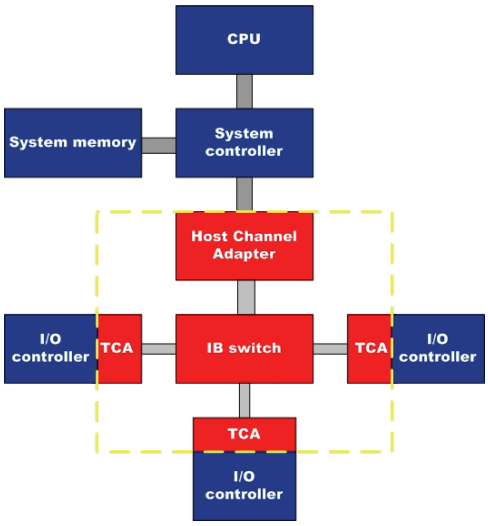


Figure 2.1: The InfiniBand architecture blends in with the rest of the hardware [Ric01]

HCA provides primary processor to the InfiniBand connection point. The InfiniBand switch provides a central connection point for multiple TCA to the HCA [Ric01].

Table 2.1 presents InfiniBand compared to other well known communication channels. It shows that InfiniBand is better in several areas.

Feature	InfiniBand	PCI - X	Fiber Channel	Gigabit Ethernet	Rapid I/O
Bandwidth (Gb/s)	4,16,48	8,51	4	1	16/32
Pin Count	4/16/48	90	4	4	40/76
Max Signal Length	Kilometers	Centimeters	Kilometers	Kilometers	Centimeters
Transport Medium	PCB, Copper, Fiber	PCB	Copper, Fiber	Copper, Fiber	PCB
Simultaneous Peer-to-Peer Connections	15 Virtual Lanes			Supported	3 Transaction Flows
Native Virtual Interface	Supported				
Multicast Support	Supported		Supported		
End-to-End Flow Control	Supported				Supported
Memory Partitioning	Supported		Supported		
Quality of Service	Supported		Supported	Supported	Supported
Reliable Connectivity	Supported		Supported		Supported
Scalable Connectivity	Supported		Supported	Supported	Supported
Maximum Packet Payload	4 KB	Bus, No Packets	2 KB	9 KB	256 B

Table 2.1: I/O Interconnect Analysis [Ric01]

PCI-X is an upgrade of the classical PCI architecture [Pen02]. There are reasons to believe that PCI-X will be the next generation of back plane connectivity for the majority of the computer users.

The way InfiniBand breaks through the bandwidth and other limitations of the PCI bus is by migrating from the traditional shared bus architecture into switched fabric architecture. Every node is connected to each other through the fabric. A node represents either a host device such as a server or an I/O device (E.g. RAID subsystem).

The fabric is a collection of interconnected switches and routers. InfiniBand can guarantee bandwidth and different service levels buy using virtual lanes [Hal02]. These are located inside the fabric and are used by the subnet managers. A subnet manager is not necessarily a separate device; it may be additional intelligence built into the subnet switch. These lanes are like multiple lanes on a freeway, dedicated for single-passenger cars and high-occupancy vehicles. The reason why the lanes are defined as virtual is because there are no physical wires inside the fabric. InfiniBand v.1.0 defines 16 virtual lanes, where 15 are dedicated to data traffic and one is dedicated to management. Since all InfiniBand devices are hot-pluggable, the fabric needs to be able to reconfigure its topology map on the fly. That is why there is a single lane dedicated only to management. The subnet managers are also responsible for negotiating and matching data rates for a point-to-point channel between two nodes. This can be needed if one node has a 4x connection and sends data to a storage system with only 1x connection. Then the subnet manager sets up a 1x connection channel without dropping packets or impeding any higher speed traffic.

2.1.3 The different layers

To explain the different functionalities of the different layers in the InfiniBand architecture, the OSI model is used (Figure 2.2). From this model the focus is on the four layers in the bottom (red boxes).These layers are the transport, network, data link and physical layer and infect the behavior of the network.

Figure 2.2: This figure shows the OSI model

Transport layer

Transport layer is responsible for the actual transportation of the packages. It controls several key aspects including packet delivery, channel multiplexing and base transport servicing. For simplicity, the majority of the transport features draw direct correlation with currently available networking technologies.

Network layer

Network layer is responsible for routing packages between the different subnets. Each package features a Global Route Header and a 128-bit IPv6 address for both source and destination nodes. This layer also provides a unique 64-bit identifier to each device in the network.

Data Link layer

At the packet communication level there are specified two distinct packet types of data transfer and network management.

- The management packages provide operational control over device enumeration, subnet directing and fault tolerance.

- Data packages contain the actual information that is transferred. Each package deploys a maximum of 4 KB of transaction information.

The link layer also allows for the Quality of Service characteristic of InfiniBand.

Physical layer

Since InfiniBand has a full duplex nature it has a requirement of only four wires, a high speed 12x implementation requires only 48 wires. This is almost just half of what is required in PCI-X. To make it more cost effective InfiniBand relies on “off the shelf” copper twisted pair and fiber optic cabling technologies. Theoretically this configuration allows multiple connections paths scaling up to 120 Gb/s in performance.

2.1.4 InfiniBand versus Ethernet

Compared to Gigabit Ethernet, the InfiniBand architecture has several advantages, especially concerning the physical layer device model. InfiniBand specifies a minimum of 0.25 watts per port for operating on copper up to a maximum distance of 16 meters. In contrast Gigabit Ethernet technology specifies a minimum of 2 watts per port and this type of connection is best utilized at distances of 100 meters or more. This difference makes it possible to develop more efficient hub design for InfiniBand than Ethernet. InfiniBand is designed especially to support low latency RDMA semantics for clustering [Lon03]. Gigabit Ethernet technology must overcome legacy issues to support such critical technology.

Gigabit Ethernet was not specially designed for clustering. By investigating the message queue depths and performance it is possible to see that it is not optimal for clustering [Web01]. InfiniBand is constructed for data center distances up to 16 meters. For longer distances, fiber is needed. With fiber it is possible to reach distances of up to 10 kilometers, but such distances give significantly higher costs. Preliminary research shows that InfiniBand has advantages compared to Ethernet in large clusters [JM03]. With clusters in the size of 32 CPU's, the use of InfiniBand will give a 40 % increased performance. Researchers emphasize that in small clusters Ethernet will give a reasonable performance and cost. In large clusters, especially those serving several users simultaneously, InfiniBand is a more scaleable and suitable solution.

2.2 RDMA

Remote Direct Memory Access (RDMA) is developed by the RDMA consortium and lets one computer directly place information into the memory of another computer. The technology reduces latency by minimizing demands on bandwidth and processing overhead.

2.2.1 Bottlenecks

In clusters and other network applications the demand for increases in network speed is growing faster than the processing power and memory bandwidth of the computer nodes that must process the network traffic [Pin02a]. As the Ethernet infrastructures are increasing in bandwidth, the CPUs on the nodes have problems sending data fast enough. RDMA moves much of the overhead of protocol processing to the Network Interface Controller (NIC) and puts incoming network packets directly into the correct destination memory location. This technique eliminates temporary memory buffering and copying associated with protocol processing.

2.2.2 Remote Direct Memory Access

Traditional copy uses the CPU to move data from one buffer to another (Marked with 1 in Figure 2.3). With DMA the CPU programs the DMA engine and the DMA engine will move the data and notify the CPU when it is done (Marked with 2 in Figure 2.3).

With RDMA the CPU can program the NIC to transfer the data. The NIC RDMA engine will access the memory directly, transfer the data, store the data directly in the remote computer memory and notify the CPU when it is done (Marked with 3 in Figure 2.3).

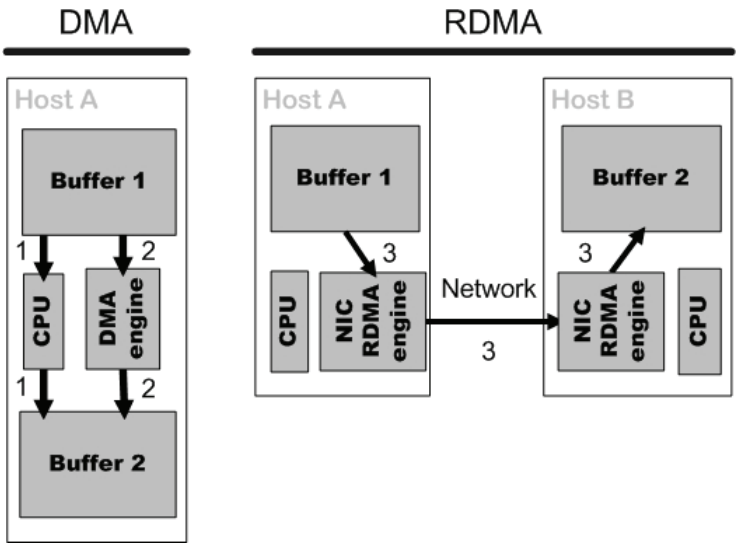


Figure 2.3: DMA and RDMA

RDMA has typically three Data Transfer Mechanisms; RDMA Write, RDMA Read and Sequence Reliable Diagrams (Sends). These mechanisms can be combined by the Upper Layer Protocol (ULP) to create ULP unique sequences that do not require the destination to process intermediate operations. Examples of ULP are NFS (Network File System) and SDP (Socket Direct Protocol).

2.2.3 SDP – An implementation of RDMA

The Sockets Direct Protocol (SDP) is an InfiniBand specific protocol defined by the Software Working Group (SWG) of the InfiniBand Trade Association (IBTA) [Pin02b][Pin03]. The SDP uses InfiniBand architecture optimized transfers while maintaining the traditional socket stream semantics. It implements the in-order delivery in hardware and the NIC demultiplexes the data stream instead of the operating system. The use of RDMA reads and writes enables direct data replacement into an application buffer. In the traditional model the kernel has three levels that the data must be processed through. This is shown to the left in Figure 2.4. The SDP bypasses the kernel by using the RDMA semantics. A RDMA enabled Network Interface Controller (RNIC) is necessary to use RDMA.

This controller can access the memory directly with operations which an RNIC Interface is expected to perform. These operations are referred to as verbs and are defined by the RDMA consortium. The network adapter vendors support the RDMA protocol by using semantics defined by these RNIC verbs. The SDP model with the kernel bypass is to the right in Figure 2.4.

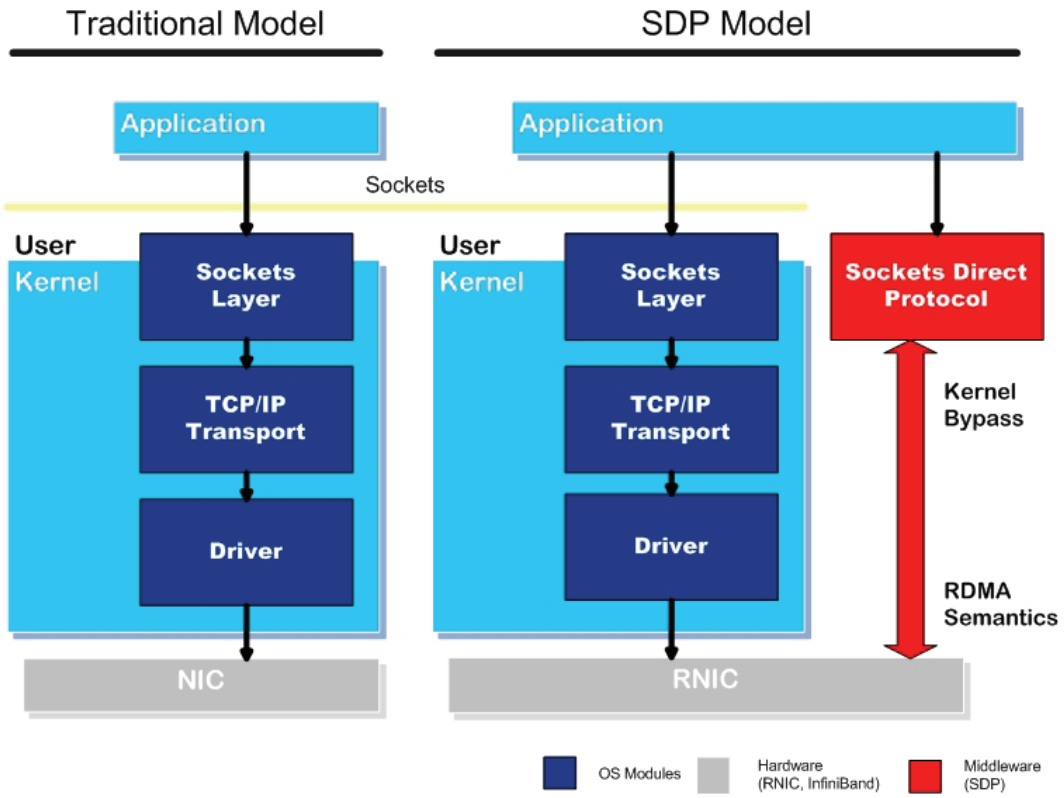


Figure 2.4: The SDP model

The SDP has two ways to copy data from the source to the receiver [Bal04+]. The first way is referred to as Zero Copy (Zcopy). If the length of the data is longer than a threshold given by the user, the data will be transferred directly from the source buffer to the receiver buffer (ULP buffers). The overhead in Zcopy is associated with pinning the ULP buffer in memory, advertising the buffer to the Remote Peer, and then transferring the data. The second copy method is referred to as Buffer Copy (Bcopy) and this method transfers the data through intermediate private buffers (SPD private buffer pool). The overhead associated with Bcopy is associated with copying the data into the send SDP private buffer pool, sending it directly to the receiver's SDP private buffer pool, and then copying it into the ULP buffer. SDP private buffer pool can also be used if the application requires it. Some applications require the network to buffer the data for good performance.

A mix of Sequenced Reliable Datagram's (Sends), RDMA read and RDMA write is used to transfer ULP data. Zcopy uses RDMA reads or writes, transferring data between RDMA buffers and Bcopy uses sends, transferring data between send and receive private buffers. The buffering model is shown in Figure 2.5.

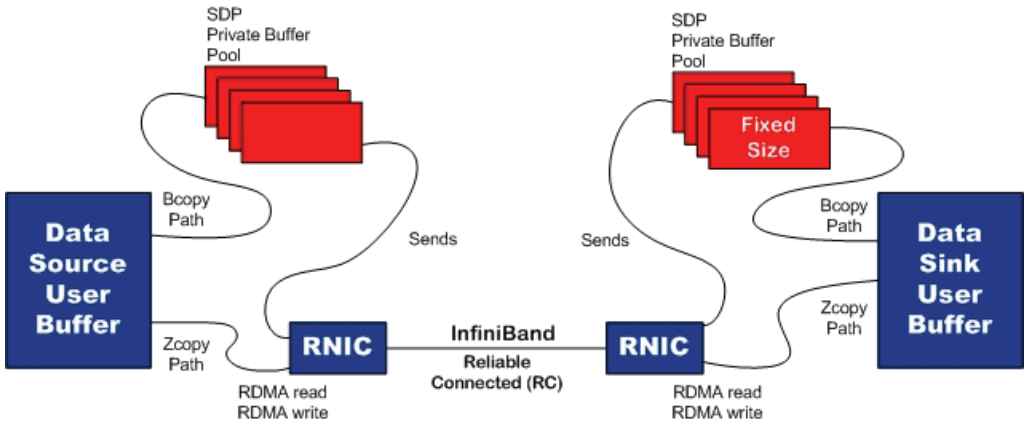


Figure 2.5: SDP Buffering Model

Figure 2.5 shows that the SDP Buffering Model has two types of buffers. The SDP Private Buffer is used for transmission of all SDP messages and ULP data that is to be copied into the receive ULP buffer. User buffers are intended to be accessed directly from the source's ULP buffer to the receiver's ULP buffer. Bcopy uses a flow control mechanism similar to the TCP Sliding Windows protocol (see glossary). It keeps sending data till the window is full

When the application reads data from the socket buffer, the receiver sends a control message back to the data source updating its windows size. Normally, Bcopy is used when the ULP buffer is smaller than the Bcopy threshold, but applications can also force all transfers to use Bcopy. This is also known as buffered mode in SDP. The Bcopy transfer mechanism is show in Figure 2.6.

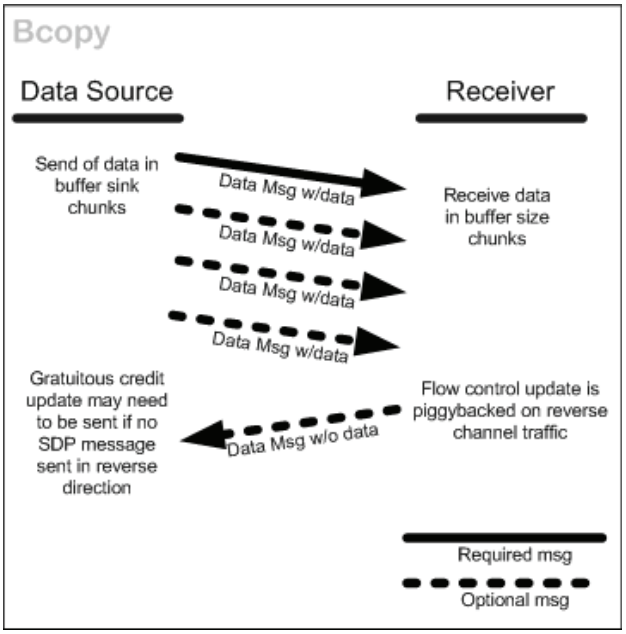


Figure 2.6: Bcopy data transfer mechanism

SDP has also two additional control messages which are used with Read Zcopy and Write Zcopy. The first is SinkAvail message which allows destination to RDMA Read from source (Sink represents the receiver). The second is SrcAvail message, which allows source to RDMA Write to the destination. These messages are known as Buffer Availability Notification. The SinkAvail message is used if the receiver already has posted a receive buffer and the data source has not sent the data message yet. This is done by first registering the receive user-buffer (for large message reads). Then a SinkAvail message is sent containing the receive buffer handle to the source. The data source on a data transmit call uses this receive buffer handle to directly RDMA write the data into the receive buffer. The SrcAvail message is used if the data source has already posted a send buffer and the available SDP window is not large enough to contain the buffer. This is done by first registering the source user-buffer (for large message sends). Then a SrcAvail message is sent containing the transmit buffer handle to the receiver. The data on a data receive call uses this transmit buffer handle to directly RDMA read the data into the receive buffer.

Figure 2.7 shows the Read and Write Zcopy mechanisms and how they use the Buffer Availability Notification.

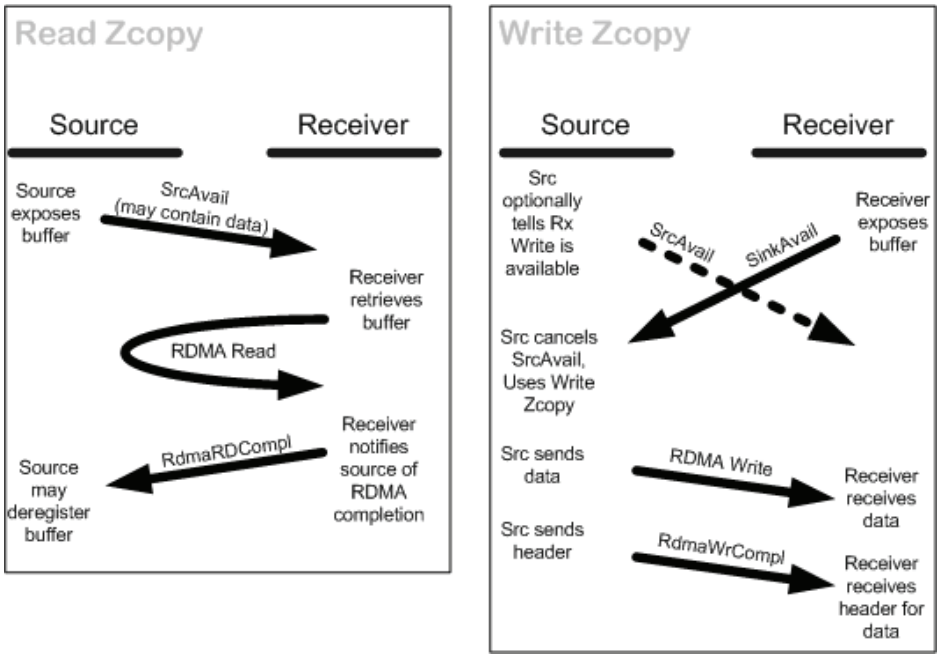


Figure 2.7: Read Zcopy and Write Zcopy. There is one less operation at Data Source when using Read Zcopy

In general, SDP combines the different copy methods in three different modes of operation:

- **Buffered mode.** If the application only wants to send a few dozen bytes, the data will be sent using send operations to the private protocol buffer (Bcopy).

- **Combined mode or transaction mode.** This mechanism use Bcopy to transfer the command and Bcopy or Zcopy for the reply. Transaction enables fewer messages and lower threshold for Zcopy.

- **Pipeline mode.** This mode can use all data transfer mechanisms (Bcopy, Read Zcopy or Write Zcopy) and the receiver can process data while the source keeps sending. It can also be used to continuously pass streaming traffic.

2.3 Copy on Write

To make highly available database systems, one of the problems to address is how to keep data available to everyone at all times and still keep the content consistent [Pet02]. Copy on Write (COW) is a technique that allows having data available at any time. Traditionally this means that one would have to operate with copies of all the data. What makes COW preferable is that it does not make this copy of the data until it is needed. This means that one only have duplicates of some of the data and therefore save space and CPU usage. In short, COW is a technique where two parties share data until one of them tries to write to it. At that point, a copy of the data is made and the two parties go their separate ways. This technique is well known from Operating Systems implementations, where it is used to optimize the file system and in virtual memory systems. When versioning or snap-shot is needed this is also a preferred technique because of the low storage overhead and the ability to version on-line. An alternative to COW for these operations is split-mirror, which copies the data blocks off-line to a new physical location on the disk. This requires a large amount of additional storage and active applications must wait while the data is copied. Although COW has many advantages it has one major disadvantage. Frequent snap-shots destroy contiguity, and therefore decrease the performance of the system. When a new allocation is made for data, it is difficult to place this near the original allocation. Over time this means that data is spread all over the disk/memory. Data that is spread all over the disk decreases the performance since the disk is a mechanical device and therefore works slowly. The idea of COW in databases is as a low-level concurrency control. Instead of using traditional locks (latching) which blocks other transactions out, COW is used to prevent transactions from waiting. When a transaction tries to update a segment locked by a read operation, the segment is copied and the transaction updates the new segment.

2.3.1 The algorithm

To give a better understanding of COW there will be an example using the string class [Gre02]. Imagine this string class implemented in such a way that it only stores a pointer to a string (Figure 2.8).

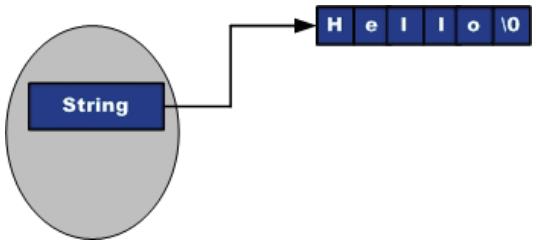


Figure 2.8: String Class with a pointer to a String

The traditional way to make a copy of such an object would be to create a new instance of the object. Then allocate enough memory to hold the string and copy the string from the old object into the newly allocated memory space. This technique is called a deep copy. If the string data is particularly large this technique would take too much time and resources. Then it would be much more effective to copy just the pointer to the string and have both instances pointing at the same string data (Figure 2.9).

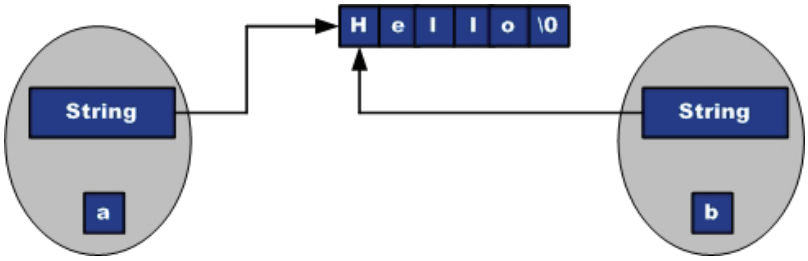


Figure 2.9: Two String Classes pointing to the same String

When using this technique the system has to make sure that if one of the string objects updates the string data the other does not get its data modified. To determine whether there is more than one string pointing to the same data a reference counter is introduced. This counter tells how many string objects are pointing to this string. When a string object wants to modify the string data it checks the reference counter. If it equals 1, the object can perform the modification to the data, but if the reference counter is greater than 1 it must perform a deep copy and decrease the reference counter. This technique, when data is not copied until it is needed, is called Copy on Write (Figure 2.10).

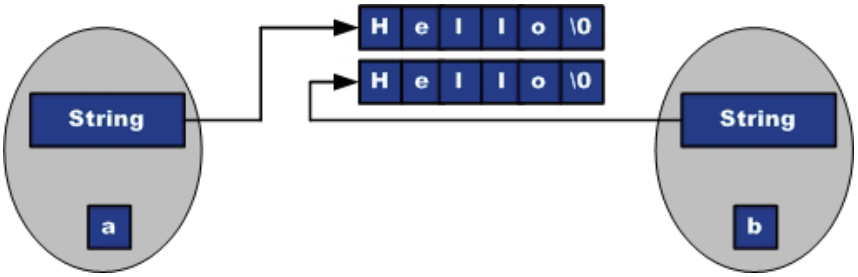


Figure 2.10: Two String Classes pointing to a copy of the same String

In this thesis segments instead of strings are used and only main memory is used for storing data. Each segment will contain a predefined number of bytes. Code 2.1 shows the pseudo code for the Copy on Write algorithm using segments.

```
if(segment locked by other process)
{
        //perform Copy on Write
        freeSegment.putSegment(lockedSegment.getSegment());
        freeSegmentPointer = lockedSegmentPointer;

        //When the locked segment is unlocked
        lockedSegment = freeSegment;
}
```

Code 2.1: Copy on Write algorithm

2.4 Checkpoint

In Data Base Management Systems (DBMS) it is necessary to have recovery mechanisms [HK04]. If a system fails, the recovery mechanism will restore it to preserve data and minimize downtime. These systems use a log to record all changes made on the system. During recovery this log is redone from the beginning to the end. To prevent this log from getting infinitely large, the system use checkpoints. A checkpoint is a consistent snapshot of the system from which this recovery can start. This checkpoint is also recorded in the log. During recovery the system typically performs three steps:

1. **Analyze the log.** The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.

2. **Redo.** Starting at the most recent checkpoint, the log is read forward and each log record is redone.

3. **Undo.** Goes backward from the end of the log and removes the effects of all uncommitted updates from the database.

These steps are shown in Figure 2.11.

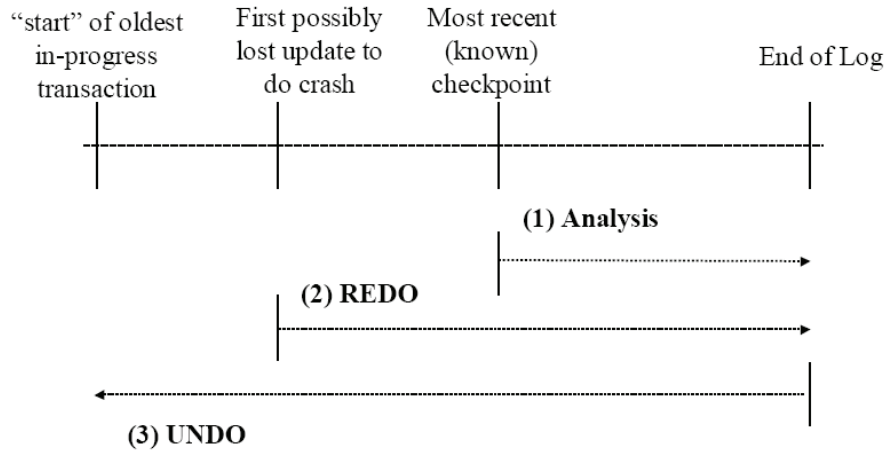


Figure 2.11: Steps during recovery [Ali01]

On a DBMS running on one computer, the system makes a checkpoint by flushing all updated data to the hard drive and recording the event in the log. Normally a fuzzy checkpoint algorithm is used to prevent a system stopping during a checkpoint.

The checkpoint process is usually run as a background process and the flushing of data to disk is done in small batches. Checkpoints can also be used in clusters and the HADB is an example of a clustered system, using checkpoints. This system has two types of checkpoint methods. The first type is internally at each node and the checkpoint is performed by flushing the data from memory to the disk. The other checkpoint method is performed between the nodes. HADB has a hot-standby for every node and they are kept synchronized by redoing the log. If a node crashes, the system finds a new node and repairs it with a checkpoint. The checkpoint transfers all the data from the primary node to the new hot-standby node. After a checkpoint the log is redone to get the hot-standby node updated.

2.5 Buffer management

When creating a main memory database one of the important aspects is how data are stored in memory. A buffer is a container of data in memory.

This container has a fixed size and contains data of a specific primitive type. In this implementation Java NIO buffer has been used. This API is most suited for this task, and explanation of buffers will therefore be focusing on Java NIO buffers.

2.5.1 Java NIO buffer

A buffer is an object which holds data that is to be written to or that has just been read [Tra03]. A buffer is essentially an array. Generally, it is an array of bytes, but other kind of arrays can be used. What separates a buffer from an array is that it provides structured access to data and keeps track of the system's read/write processes. All NIO buffers contain three state variables and these are used to read/write data from/to the buffer. Together these variables track the state of the buffer and the data it contains. These variables are (Figure 2.12):

- **Position.** The position keeps track of how much data that has been written. It specifies into which array element the next byte will go. If 5 bytes are read into the buffer, then the position will be set to 5 pointing to the sixth element in the array. Likewise, if there is writing from the buffer. Position keeps track over how much data that has been received from the buffer. If 2 bytes are written to a channel the position will be set to 2 referring to the third element in the array.

- **Limit.** This variable specifies how much data there is left to get when writing from the buffer to a channel. When data is read into the buffer, the limit tells how much room there is left in the buffer. That's why: position ≤ limit

- **Capacity.** Capacity specifies the maximum amount of data that can be stored in the buffer. It specifies the size of the underlying array, or at least the amount of data allowed using.

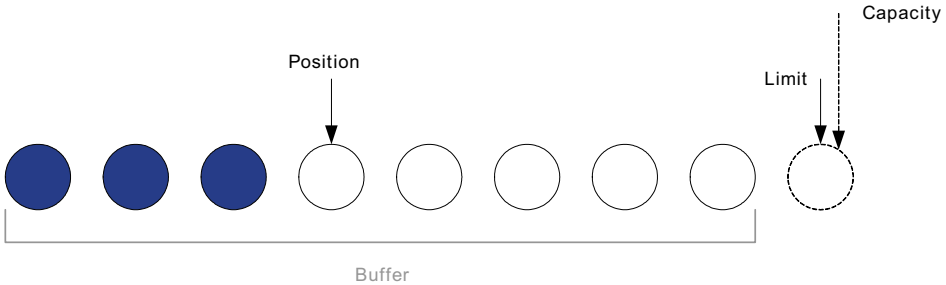


Figure 2.12: State variables

2.5.2 Using byte buffers

When using buffers in Java NIO one needs to allocate memory space before one starts working. In NIO the method *ByteBuffer.allocate(size)* can be used. This method allocates the underlying array of the specified size and wraps it in a buffer object.

If this is done manually it will be the same as using these commandos (Code 2.2):

```
Byte array[] = new byte[1024];
ByteBuffer buffer = ByteBuffer.wrap(array);
```

Code 2.2: Commandos in ByteBuffer

When using byte buffers in Java NIO, there are two choices; direct or non-direct buffers [SMI04]. In this thesis direct byte buffers are used. Direct buffers are allocated in a special way in memory to increase bandwidth. Sun's definition gives a better understanding of what a direct buffer is:

Given a direct byte buffer, the Java virtual machine will make a best effort to perform native I/O operations directly upon it. That is, it will attempt to avoid copying the buffer's content to (or from) an intermediate buffer before (or after) each invocation of one of the underlying operating system's native I/O operations.

Allocating and de-allocating direct byte buffers have higher administrative costs than non-direct buffers. It is recommended for direct buffers to be allocated primarily for large, long-lived buffers that are subject to the underlying system's native I/O operations. Since the content of these buffers may reside in the normal garbage-collection heap, their impact upon the memory footprint of an application might not be obvious.

A non-direct buffer is allocated logically to store data. This means the system does not need to worry about memory allocation [Cha02]. Non-direct buffers are spread all over the memory, while direct buffers are allocated in a contiguous memory block. This can be seen illustrated in Figure 2.13.

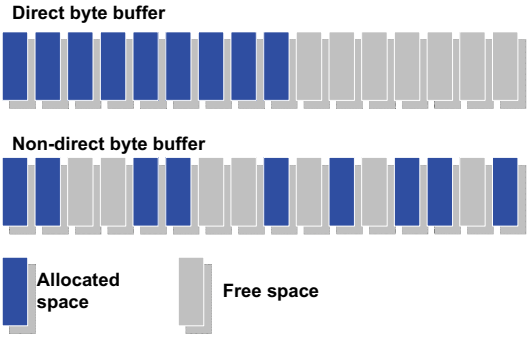


Figure 2.13: Direct vs. Non-direct byte buffer

Figure 2.14 shows how Java native works. This technique is used when direct buffer is implemented. The main issue is the connection with the host operating system. This operation increases the I/O, but is costly in the meaning of CPU usage.

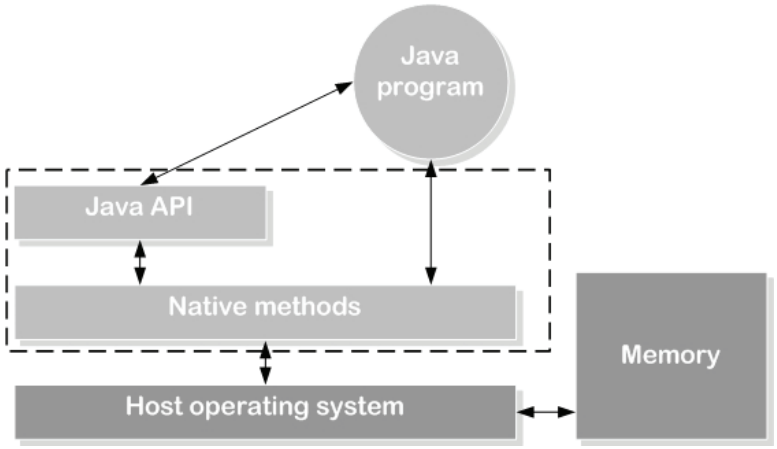


Figure 2.14: How Java native works (Direct buffer)

Byte buffers are one of the technology components used in the simulator. In addition to technology, the preliminary experiences are important to create the most realistically behaviors of the components. E.g. *what is a realistic bandwidth of the InfiniBand?* Chapter 3 describes these experiences and summaries the settings that will be used.

Chapter 3

Performance of InfiniBand/RDMA

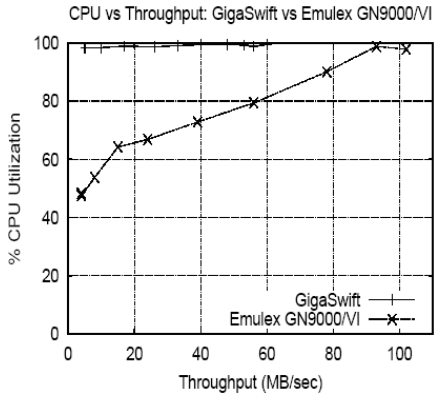
When creating a simulator like the one in this master thesis, it is very important to find relations to earlier research done on the subject. By studying benchmarks and identifying important factors, the simulator is given the optimal settings and behavior. The use of InfiniBand the way it is intended in this thesis is a new field for research. The approach used to verify the data is that single relevant subjects from earlier research are taken out and used as guidelines when creating and calibrating the simulator. By doing this it will be much easier to predict the results from the simulator. This theoretical background will also be very important when setting up hypotheses for the test cases.

In this chapter the most interesting articles are presented. These articles give different views of the use of InfiniBand and provide many guidelines. These guidelines have been used during the development of the simulator. Even though none of the articles describe a system similar to the one developed in this thesis, is has been possible to see relations in many areas.

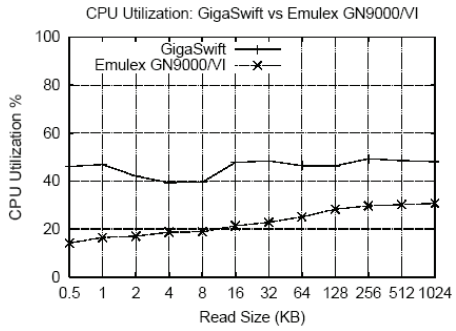
3.1 CPU usage

Conventional TCP/IP communication is a costly method of copying data between kernel buffers and user process virtual memory at the socket layer. With RDMA the kernel bypass reduces processing overhead and thus is should reduce the CPU usage. In Callaghan el al. [Cal03+] RDMA has been used as an RPC (Remote Procedure Call) transport layer. NFS has been run with this new transport technique and compared with traditional transport techniques like Gigabit Ethernet. The article shows results from an implementation on two Sunblade 1000 machines. These two machines use two connections.

One connection used Sun GigaSwift Ethernet adapters and the other used Emulex GN9000/VI adapters for RDMA traffic. In both of these drivers the use of Jumbo frames was enabled. Jumbo frames extend the normal Ethernet MTU (Maximum Transmission Unit) from 1500 bytes up to 9000 bytes. With normal frames the performance dropped 40 % on the Emulex GN9000/VI. A benchmark program was used to measure the CPU usage. This program read a file of 1 GB sequentially and could vary read ahead from 0 to 16 reads ahead. The results were plotted with CPU usage versus throughput. The plot (Graph 3.1) shows that reading the file over GigaSwift, even at low throughput from smaller reads, requires almost all of the CPU, while at similar throughput the Emulex RDMA requires significantly less CPU. The authors suspected that much of the reduction in CPU use is due to off-load of network processing into the Emulex CPU. This test was also run with no read ahead to eliminate the effect of asynchronous read ahead threads. With this configuration, the plots (Graph 3.2) showed that Emulex RDMA is using between 30 % and 60 % less of the CPU than the GigaSwift uses for the same NFS transfer size. Another study of CPU usage was done in 2002, by researchers [SMS02] who did a study of the impact from use of direct access I/O on DBMS. This was done by evaluating impact experimentally. A host server which provided the client API was connected to an I/O server, which provided the data, with Giganet's cLAN implementation of the VI Architecture. Each server had four 400MHz Pentium II Xeon processors, 1 GB RAM and two 10,000-RPM hot swappable SCSI disks.

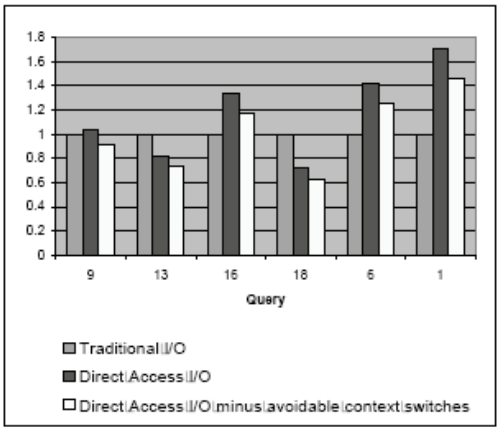


Graph 3.1: CPU vs. Throughput [Cal03+]



Graph 3.2: CPU Utilization: no read-ahead [Cal03+]

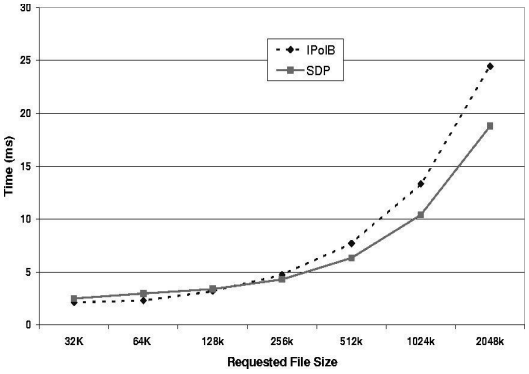
The DBMS (in this case, DB2) performs I/O by using a direct access API at the host server (RDMA). Some changes had to be made in the DB2 v7.1 software to enable it to perform direct access I/O. Each thread connected to the remote I/O server. All read and write calls required that the memory they had accessed to be registered and all file system calls were replaced with calls to the direct access I/O client library. The direct access I/O client library was also modified to support multiple threads and processes, as well as to provide sufficient functionality to support a DBMS. The experiment used five runs of six different TPC-H queries and the results were plotted in histograms that were relative to the old system. The authors expected to see a significant reduction in the number of context switches performed on the host system, because a transition to the kernel and back can be avoided with direct access I/O. The results did not show this trend, but gave an increase in the relative number of context switches. The CPU usage (Graph 3.3) also showed the same trend and the relative process privileged time. This was the amount of time the kernel spent executing work for the monitored process. It showed that small queries use more CPU time than traditional I/O. With larger query the CPU usage decreased. The authors found that this was due to an extra amount of context switching. When this was avoided, the performance was achieved with direct I/O for data - intensive queries. The conclusion was that the increased CPU usage was due to administrative costs and a poor structure in the system. Depending on the implementation, the modification of existing programs to do direct access I/O may not be trivial. Given a suitable system, direct access I/O still has good potential to reduce the amount of CPU usage by a DBMS using traditional I/O.



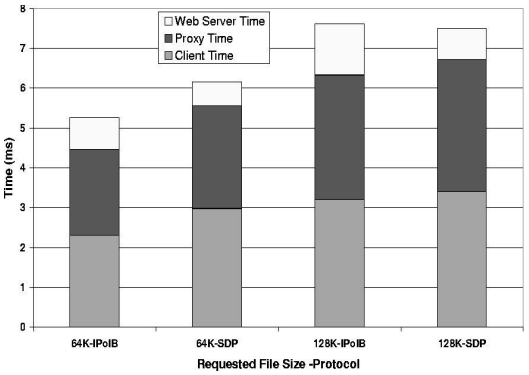
Graph 3.3: Process Privileged Time per I/O (relative to Traditional I/O) [SMS02]

3.2 Response time

D. K. Panda wrote in 2004 an article about design of scaleable data-centers with InfiniBand [Pan04]. Since electronic interaction and communication have been an important point for many companies, highly scalable, highly available and high performance web servers have been a critical part of many companies' infrastructure. To provide such services the use of data-centers has been a central requirement. The goal of this article was to take advantage in the recent improvements in network technology, such as InfiniBand to satisfy such requirements and solve the common problems encountered in a large scale data-center. The test bed used in the article consists of a cluster with 16 nodes with proxy servers using Apache or Squid, web servers using Apache 2.0, application servers using PHP, database servers using IBM DB2 and MySQL. The article compares the response time for SDP with native sockets implementation over InfiniBand (IPoIB). The results show a significantly better performance in response time for SDP over IPoIB using a message size over 128K. The difference can be seen in Graph 3.4. To understand the lack of performance benefits from small file sizes, the response time need to be split up. In this experiment the response time consists of three parts; web server time, proxy time and client time. Though the web server time reduces significantly from IPoIB to SDP, the time taken at the proxy server is higher. This is because the proxy uses a significant longer time to connect to the back-end server when using SDP. But this connection time is fairly constant and therefore becomes a smaller part of the response time as the message size increases.

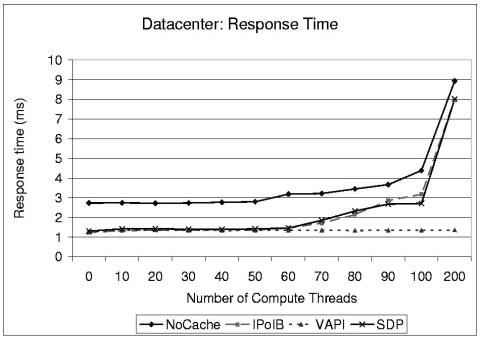


Graph 3.4: Response time [Pan04]



Graph 3.5: Response time split up [Pan04]

SDP has a 500 μs longer connection setup time than IPoIB. For the further work in this thesis, 500 μs has been chosen to be an approximation connection setup time in the simulator. The simulator will not run in a web server environment, but after discussion with all the involved parts this value is found feasible for the purpose. The article also shows that SDP and VAPI have a significant better response time than IPoIB when using active caches in data-centers. The difference between the native InfiniBand Verbs Layer (VAPI), SDP and IPoIB is significant when the number of compute threads exceeds 70. While VAPI stays stabile, SDP and IPoIB increase their response time significantly (Graph 3.6).

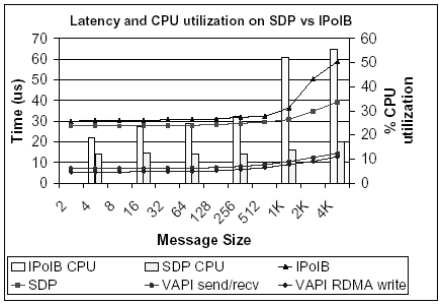


Graph 3.6: Response time [Pan04]

3.3 Latency

In 2003 P. Balaji et al. wrote an article called “Sockets Direct Protocol over InfiniBand in Clusters, Is it beneficial?” [Bal03+]. This article compares the SDP protocol with IPoIB and VAPI to find out which protocol gives the best performance. The tests were performed on two different software infrastructures: Multi-Tier Data Center environment and the Parallel Virtual File System. The tests were performed on an 8 node cluster built around SuperMicro SUPER P4DL6 motherboards and GC chipsets which include 64-bit 133 MHz PCI-X interfaces. Each node had two Intel Xeon 2.4 GHz processors with a 512 KB L2 cache and a 400 MHz front side bus. The machines were connected by a Mellanox InfiniHost MT23108 DualPort 4x HCA adapter through an InfiniScale MT43132 Eight 4x Port InfiniBand Switch. The SDK version is thca-x86-0.2.0-build-001 and Linux RedHat 7.2 operating system was used.

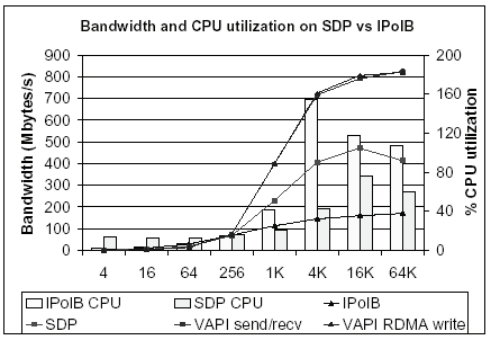
One of the subjects tested was the latency. Latency is the transit time through a digital process, from input to output. It is a minimal, and usually undesirable, delay [Etv04]. This was compared between all the three protocols together with the CPU usage. As can be seen from the graph below, the VAPI protocol is superior to the others. The graph also shows a significant deterioration of the IPoIB protocol when the message size exceeds 1KB. The trends showed in this graph demonstrate that for messages over 1K the IPoIB is not a viable alternative if a good response time is an important factor in the system. Even for small message sizes, VAPI gives a better performance with a factor of up to 5.46 compared to the IPoIB. In network technology latency is a very relevant subject. In this thesis the latency has been integrated in a bandwidth of 8 Gb/s.



Graph 3.7: Latency [Bal03+]

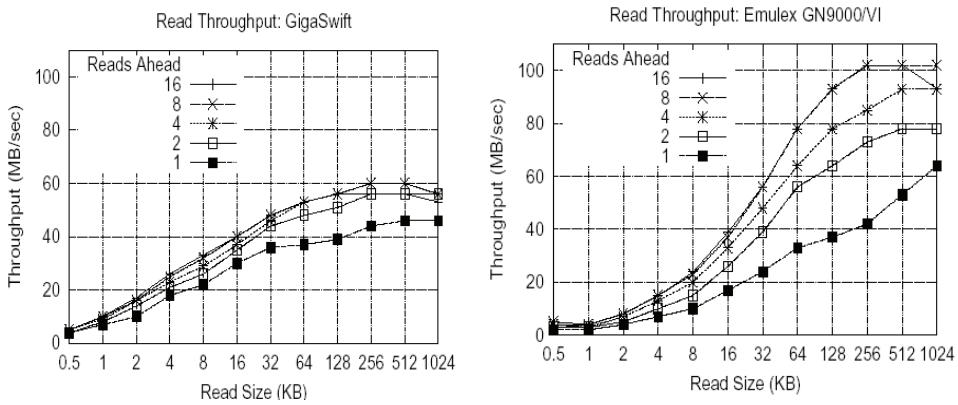
3.4 Bandwidth

In the article written by P. Balaji et al. bandwidth is also evaluated. The bandwidth shows how much data each protocol can transfer from one place to another. When comparing VAPI, SDP and IPoIB the results are clear. The VAPI outperforms both SDP and IPoIB for messages larger than 256 B. At a message size of 64 KB, VAPI transfers 825 MB/s and SDP 471 MB/s, compared with the IPoIB that transfers 169 MB/s. The main reason for choosing InfiniBand is mostly that this will give a good bandwidth, but as can be seen in Graph 3.8, this bandwidth also depends on the protocol implemented in the system.



Graph 3.8: Bandwidth [Bal03+]

The VAPI gives an improved performance by a factor of up to 4.8 compared to IPoIB, while SDP gives an improved performance by a factor of up to 2.7 compared to IPoIB. In the article written by B. Callaghan et al. the bandwidth is also evaluated (Graph 3.9). In this article Ethernet was used for network connections. These experiments show that the use of RDMA gives an advantage of over 60 % with 102 MB/s compared with regular Ethernet technology.



Graph 3.9: Throughput GigaSwift to the left and throughput Emulex to the right [Cal02]

3.5 Summary of RDMA performance

For this master thesis some of the parameters from these previous chapters will be used in the development of the simulator. This chapter gives a summary of these parameters. Chapter 3.1 shows that RDMA needs less CPU than regular Ethernet technology when reading data. This also applies when read ahead is turned off. Even though this seem to show a clear trend, the chapter also show an example of an experiment which demonstrate that CPU usage can increase when using direct I/O compared to traditional I/O. This was probably due to the administrative costs caused by poor implementation of the system. It is likely that RDMA will use less CPU than regular Ethernet technology when implemented properly.

When it comes to response time, Chapter 3.2 shows that SDP gives very good response times for messages over 128 KB. This chapter also shows that SDP has a 500 μs longer connection time than IPoIB due to the administrative costs associated with managing memory mapping. Since IPoIB does not have these administrative costs, this value is believed to be the setup connection time for RDMA.

Chapter 3.2 also shows that Native InfiniBand Verbs Layers give good performance when many compute threads are introduced. The research shows that RDMA will have a long connection time for small messages. When messages increase this connection time will almost disappear and the technology is very scalable especially when a large amount of compute threads are introduced. The problem concerning long connection time is also shown in Chapter 3.3.

Chapter 3.4 compares the bandwidth when using different network technologies. Here the VAPI shows an improved performance by a factor of up to 4.8 compared to IPoIB. And when comparing RDMA over Ethernet with traditional Ethernet technology, RDMA shows an advantage of 60 % with 102 MB/s. When using RDMA over InfiniBand, it should be possible to get a very good bandwidth compared with existing technologies. In this master thesis it is set to 8 Gb/s (1024 MB/s) and represents an throughput which is more likely to achieve since other factors in the system may reduce the throughput; e.g. bad implementation and algorithms optimized for other network technology.

Chapter 4

Hypotheses

From the previous chapters it is possible to put forward some hypotheses. These hypotheses are behaviors and trends that are reasonable to believe will appear, based on the theoretical information presented in Chapters 2 and 3. The hypotheses were introduces to represent clear evaluation goals for the simulation. Oncoming tests will either prove or disprove the hypotheses.

4.1 The hypotheses

I. If the segment size increases, then the TPS will fall. As the number of segments decrease, the probability to perform Copy on Write will also increase. The time and resources it takes to perform this operation will also increase, since there is more data to copy due to lager segments. This will result in lower performance.

II. If the segment size increases, then the model will show better throughput (MB/s) for RDMA. This is because the setup connection time for RDMA will be a smaller percentage of the total checkpoint time as the segment size gets larger. It will therefore be possible to transmit more data per second.

III. The transfer rate (bandwidth) inside the main memory will stay the same. There is nothing in this model which should affect the main memory performance. The load on the system will be constant during the whole test. Therefore, the transfer rate should stay stable at the bandwidth specified by the hardware (approximately 2 GB in this thesis).

IV. If the segment size increases, then the number of Copy on Write operations will increase. There will be fewer segments and the probability for a client of hitting a segment locket by the checkpoint process will increase.

V. If the segment size increases, then the CPU usage will also increase. Due to the increased amount of bytes to copy and number of Copy on Write operations, more CPU resources will be used.

VI. If the number of Copy on Writes increases, then the number of checkpoints will also increase. The total run-time for the program will also increase and therefore there will be more time to take checkpoints.

VII. If the bandwidth (throughput) increases and the number of segments decrease, then the checkpoint time will decrease. Thus, it will take a shorter time to transmit the same amount of data. Repair time will asymptotically draw nearer the lower limit given by the bandwidth.

4.2 Validation of the hypotheses

The hypotheses will be tested by simulating a DBMS using InfiniBand and RDMA for repair with checkpoints. This simulation consists of three steps; modeling, programming and performing test cases. The model sets the framework for the simulator and defines the behavior of InfiniBand, RDMA and a highly available DBMS. This model is designed based on theory presented in Chapters 2 and 3, which gives a good understanding of how RDMA over InfiniBand works. The framework is then used to develop a program with InfiniBand and RDMA settings based on Chapter 3. When the software is developed, tested and found valid, the simulation begins. The simulation consists of several test cases with different parameter settings. The test case results will be the basis for evaluating the hypotheses. If some of the hypotheses fail, it is important to find the solution as to why this happened, to eliminate possible errors in the model. A summary of the results from the hypotheses can be found in Chapter 7.9.

Chapter 5

Design of the simulator

The motivation for developing this software was to simulate checkpointing of a relation database over InfiniBand. The database system had to be implemented in a simple way because the limitations of the thesis's scope, but it were important that the simulator was created with the same behavior as if it was a complete database system. The system consists of a database, which is accessed by a checkpointer and by several transactions, also called client threads. The transactions update the database while the checkpointer transfers portions (also known as segments) of the database over InfiniBand using RDMA. This will simulate a repair operation of a hot-standby node in HADB. In the implementation of the simulation there are two issues categorized as the most important and the most difficult parts to implement. These are the transactions and the RDMA processes. The problem with the implementation of the transactions is that it is very hard to implement a realistic transaction load. The difficulty of implementing RDMA arises from the issue that this research subject uses state-of-the-art technology, which means there has been little preliminary research on the subject. That is why the RDMA function is assumed to be a linear scaling of the transfer rate as function of the segment size transferred. The connection setup time is assumed to be 500 µs. This is based on preliminary research which shows that SDP over InfiniBand has a connection time which is 500 µs longer than IP over InfiniBand (Chapter 3.2). The system simulates one node in a database cluster and the database refers to a fragment of the whole database.

During development of the simulator an iterative development process was used. The process consisted of several activities. The activities can be divided into three parts; design, implementation and test. In parallel with these activities, a continual update of the thesis report has been done.

These three parts were performed in an iterative way. Iterative work means that the different phases are processed with a different accuracy. In new terminology this kind of development is also called agile software development. When developing software using agile development methodology it is possible to make changes to the design even late in the project. This makes the whole project much less vulnerable to changes and the chances of success increase considerably. The disadvantage is that the solution may become too general and contain unnecessary flexibility. For more detailed information about the development process, see Appendix I.

5.1 The simulator

The program consists of three main parts; the database which has all the data, transactions that do operations on the data and the checkpoint, which sends the data over the network. In Figure 5.1 the relation between these three parts can be seen. Every part is initialized by the *DataBaseControl* class.

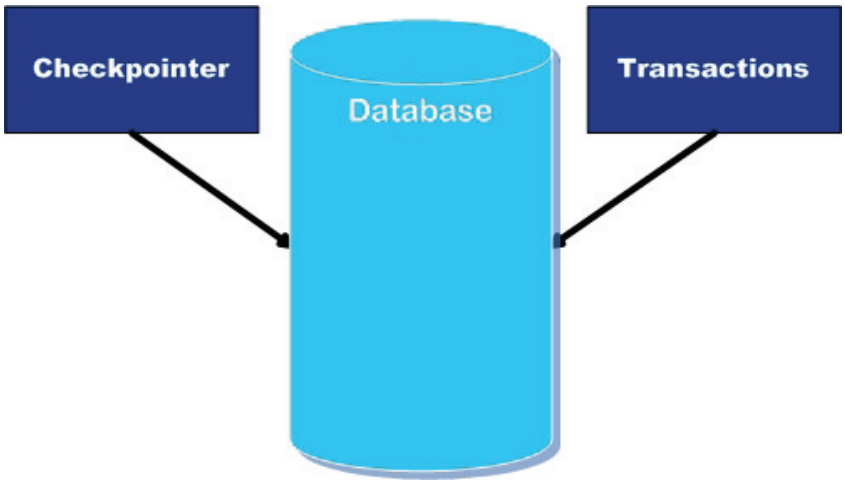


Figure 5.1: The three main parts in the program

The *DataBaseControl* creates the *BufferManager* object which contains the database, executes the *TransactionController* thread to control the transactions and *Checkpoint* thread to frequently perform checkpoints. The monitor processes uses *accept()* and *addResults()* in *DataBaseControl* to save results from monitoring. The *accept()* method records CPU usage and the *addResults()* records time spent by the different *ClientThread* objects. The classes are shown in Figure 5.2 and the complete source code can be found in Appendix II.

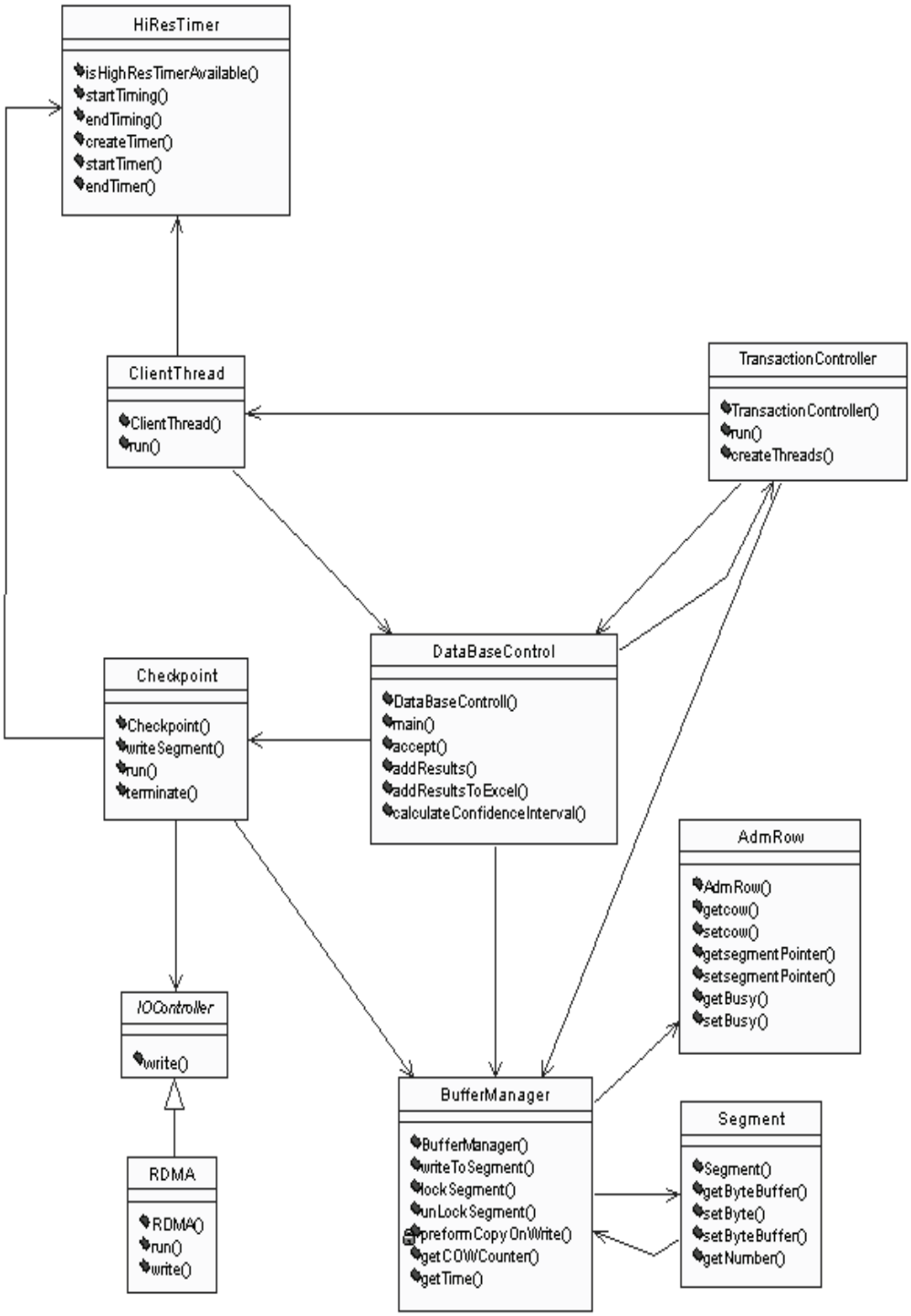


Figure 5.2: Class diagram

BufferManager creates the database, segment table and administrative table (Figure 5.3). The *database[]* contains the data and the *admRow[]* contains information about the database. One object in the *admRow[]* corresponds to one segment in the database. The *Segment[]* table contains *Segment* objects which are separately allocated spaces in memory and contains methods for writing and reading bytes from this space. The *AdmRow* class collects information about concurrency control, pointers and changes on one segment. This system uses Copy on Write for low-level concurrency control. *AdmRow* has *getSegmentPointer()* and *setSegmentPointer()* methods to record and read information in an *AdmRow* object. The transactions use the *writeToSegment()* method in *BufferManager* to write information to the database. This method will be explained in Chapter 5.1.2. The checkpoint uses *readSegment()*, *lockSegment()* and *unlockSegment()* during an execution. The transactions are controlled by the *TransactionController* class, Figure 5.4.

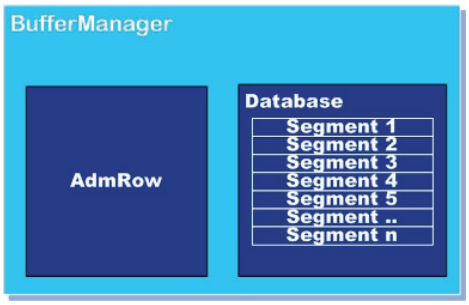


Figure 5.3: Database structure

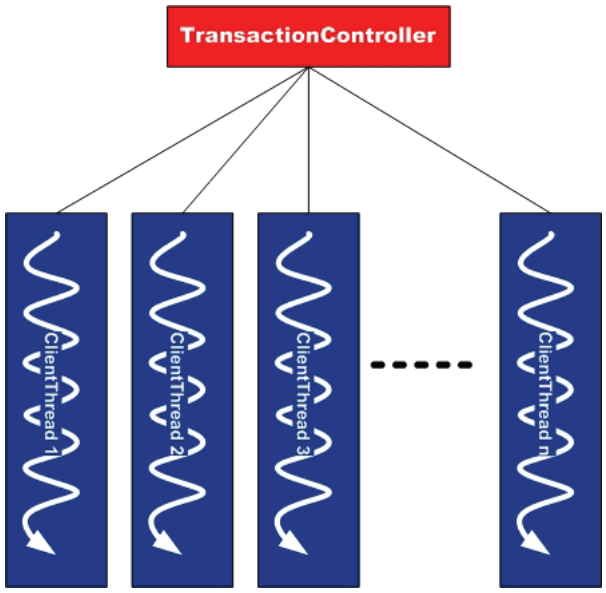


Figure 5.4: Transaction controller

This class is executed by the *DataBaseControl* class and creates multiple threads of the *ClientThread* class. One *ClientThread* can perform many transactions sequentially on the database. Checkpoints are performed frequently and send the data over the network by RDMA.

The *Checkpoint* class is executed by the *DataBaseControl* class and operates on one segment at a time (Figure 5.5). When a segment is sent over the network, the checkpoint locks it by setting a Copy on Write flag in the *BufferManager*, sends the segment address information to *RDMA* object and unsets the Copy on Write flag in the *BufferManager* again. The *RDMA* class simulates the transfer over the network, by calculating and waiting the time spent by a real network.

Figure 5.5: Checkpointer

5.1.1 DataBaseControl

The *DataBaseControl* class is the main class. This is where everything is started and controlled. When starting the *DataBaseControl* the command *java -XX:MaxDirectMemorySize=1G DataBaseControl segmentSize* is used to allocate such amount of the memory needed to run this program. The class defines all the different objects and sends references to these objects as a parameter when creating other objects. This ensures that there is just one instance of each object. In the start of this class all the parameters from the properties file are loaded into the system. The parameters that are loaded are:

- **Logging.** This parameter turns on and off debug logging.
- **MaximumFractionDigits.** This parameter controls the resolution on the CPU usage parameter.
- **DBSizeKB**. This parameter sets the size of the database
- **numberOfTransactionThreads**. This parameter controls the number of *ClientThreads*. E.g. how many threads that will work in parallel.
- **numberOfOperationsPerTransactionThread**. This parameter controls how many operations each thread is supposed to perform. This determines how long each thread will run.
- **Load**. This integer turns on and off the distorted transaction load. 0 means that uniform load is used, 1 means that distorted load is used. Distorted load is explained in Chapter 5.1.3

After all the different objects are created the *DataBaseControl* starts the different threads by using the command *Thread.start()*. Both the *TransactionController* and the *Checkpoint* object are defined as threads and are run completely independently from the rest of the system. To store information about every run a result file is created (Figure 5.6). The name of this file is of the form *"hh-mm-ss-dd-MM-yyyy.res" (hour-minute-second-day-month-year)*. This file contains different information about the data collected during performance and looks like this.

```
DATABASE INFORMATION
*****************************************************************
Database size:                 1000000 Kilobyte
Segment size:                  14600 Kilobyte
Simultaneous threads:          10
Operations per thread:         15000

RESULTS
*****************************************************************
The program used:              29361.183527545196 milliseconds
TPS:                           5114.479137880418
Number of checkpoints:         15
Number of CopyOnWrite:         2457
Average COW time:              5.6324013101003321
Average checkpoint time:       14.725449776011947 milliseconds
Average CPU usage:             0.5370756482224004 %
```

Figure 5.6: Result file

The information under “Database Information” is collected straight from startup parameters. This information is stored because this makes it easier to look at old files when all the active parameters during this particular run are known. The results are calculated before they are written to file. For every thread and interesting function in the system there is a timer. Timing data together with a set of different counters makes it possible to calculate different results. The results from the timers are made available for the *DataBaseControl* object when the different threads call the function *addResult()* or *addResultToExcel()* in *DataBaseControl* and the values are added to different variables.

At the end the following results are calculated:

$$TPS = \frac{1000 * NumberOfClientThreads}{AverageClientThreadRuntime * RoundsPerClientThread}$$

$$Averagecheckpointtime = \frac{TotalTimeOfAllTheCheckpoints}{NumberOfCheckpoints}$$

$$AverageCPUUsage = \frac{AllTheCPUUsageSamplesAddedTogether}{NumberOfCPUSamples}$$

$$AverageCOWTime = \frac{\sum COWTimes}{numberOfCOWOperations}$$

In addition to this file there is also defined a result file, which contains all these results in a formatted way to make it possible for import to MS Excel (Appendix III). This result file just adds new information for each run. This makes it possible to run the program several times and collect a lot of results before importing to MS Excel. The *DataBaseControl* also contains the method *accept()* (Code 5.1) which is called from the *CPUUsage* native code. The parameter to this function is the sample result from the *CPUUsage* class.

```
public void accept(final SystemInformation.CPUUsageSnapshot event)
{
       if (m_prevSnapshot != null && !stopPrint)
       {
              CPUResult+=((100.0
              *SystemInformation.getProcessCPUUsage(m_prevSnapshot,event)));

              CPUCounter++;
       }
       m_prevSnapshot = event;
 }
```

Code 5.1: CPU usage method

The *DataBaseControl* creates a lot of objects, one of these being the *Logger* object. This object refers to the logger class which is used to create a debug log. This log is used to follow the dataflow in the system. This kind of testing is also called white box testing.

The debug logging can be turned on and off by the *logging* parameter in the properties file. To log special events in the system the *LoggerObject.logMessage(message)* is used.

The method *addResultsToExcel()* (Code 5.2) is employed to add results which are used to calculate different confidence intervals. *ClientThread*, *Checkpoint* and *BufferManager* add their timing results in this method. The parameter *number* is used to detect which object is trying to update its result. To avoid any bad results, this method is made synchronized. Use of synchronized methods does not affect the collected data, since the method is used after all the timers and counters are stopped.

```
public synchronized void addResultToExcel(int number, double result)
 {
     if(number ==0)
     {
         tpsRes+=result;
     }
     else if(number==1)
     {
         checkpointRes+=result;
         checkpointRes2+=result*result;
     }
     else if(number ==2)
     {
         cowRes+=result;
         cowRes2+=result*result;
     }
     else if(number ==3)
     {
         tpsRes2+=result;
     }
 }
```

Code 5.2: addResultsToExcel()

Figure 5.7 shows how the *DataBaseControl* starts three separate threads and how they live their own lives.

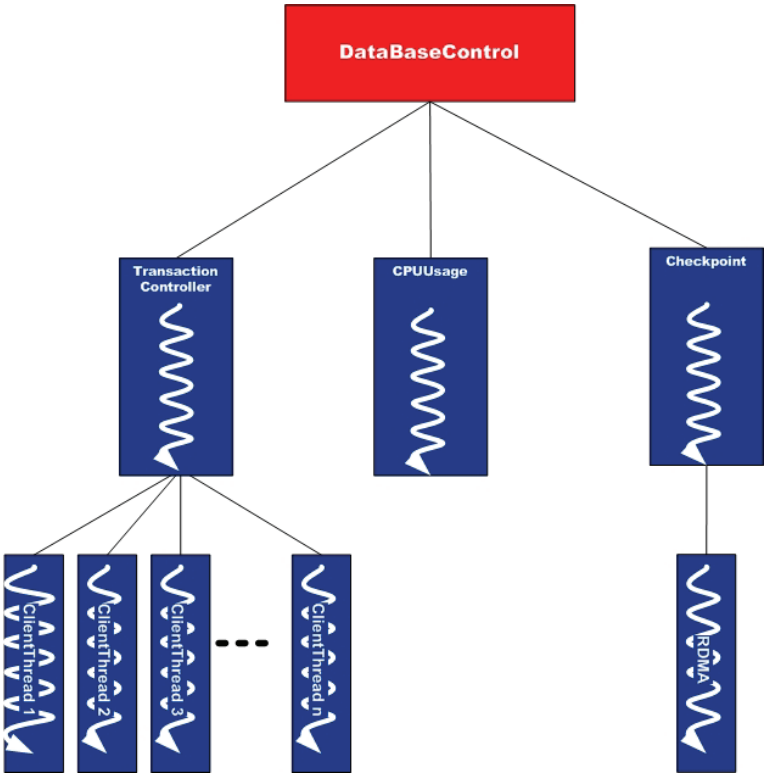


Figure 5.7: DataBaseControl as thread manager

DataBaseControl waits until *TransactionController* is done. *TransactionController* waits until all *ClientThreads* are done. Then *DataBaseControl* stops the checkpoint thread and kills the sampling of the CPU usage. To wait for a thread the *Thread.join()* command is used. This makes the current thread wait until the selected thread has finished. The sequence diagram below (Figure 5.8) shows how *DataBaseControl* communicates with the other classes. *DataBaseControl* creates and starts most of the threads. In this way, the managing of the threads becomes a lot easier.

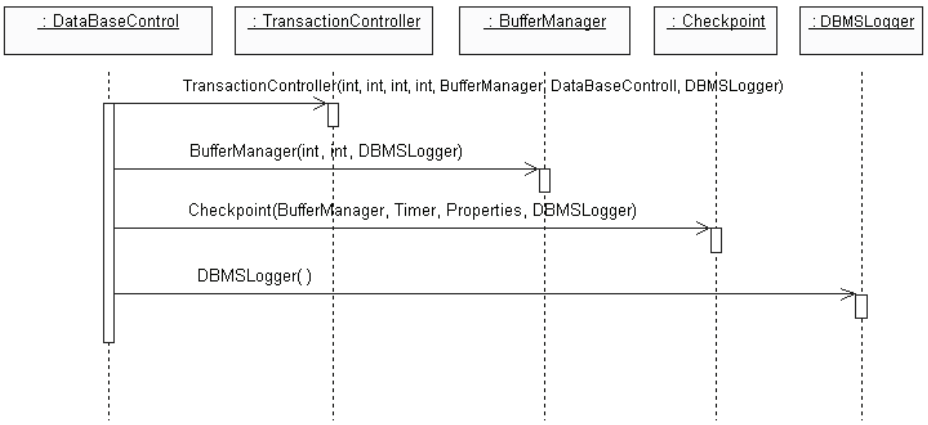


Figure 5.8: Sequence diagram DataBaseControl

The collected results are used to calculate the different confidence intervals. This is done in the method *calculateConfidenceInterval()*. The definition of confidence interval is [Woo04]:

Confidence interval (CI): The range of numerical values in which we can be confident (to a computed probability, such as 90 or 95 %) that the population value being estimated will be found. Confidence intervals indicate the strength of evidence; where confidence intervals are wide, they indicate less precise estimates of effect.

The confidence interval is calculated in two operations. The first thing that is calculated is the standard deviation; this is done by using the formula [Ras91]:

$$SD = \frac{\sum\left(x^2\right) - \sum\left(x\right)^2}{n\left(n-1\right)} \qquad (5.1)$$

SD = Standard deviation
x = checkpointtime / CopyonWritetime / Transactiontime,
n = Number of samples taken during the run

When the standard deviation is calculated, the confidence interval is calculated based on this value. The confidence interval is calculated by the following formula:

$$CI = c * \frac{SD}{\sqrt{n}} \qquad (5.2)$$

CI = Confidence interval
n = Number of samples taken during the run
SD = Standard deviation
c = Cumulative standard Gaussian distribution

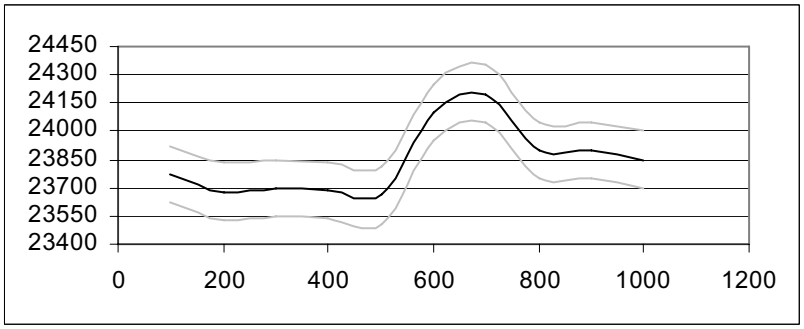
The confidence interval is calculated for checkpoint time, Copy on Write time and the TPS. These values are added to a string which is printed in the end of the resultFormatedForExcel.txt file.

To calculate the c-value, the cumulative standard Gaussian distribution is used (Equation 5.3).

$$c = GaussianTableEntry\left(1 - \frac{1 - pl}{2}\right) \qquad (5.3)$$

pl = Percentage level of confidence interval

Applying this data to one of the results gives a graph similar to the one below. Here the confidence interval can be seen as grey lines on both sides of the black line (Graph 5.1).



Graph 5.1: Example of confidence interval

5.1.2 BufferManager, Segment and AdmRow

This is the control class for the database. This class is responsible for maintaining all the necessary information about the database. When a *ClientThread* or *Checkpoint* wants read or write to the database they use methods in this class. In other words, *BufferManager* creates an interface between the database and the user. Behind the *BufferManager* there is architecture similar to the one shown in Figure 5.9.

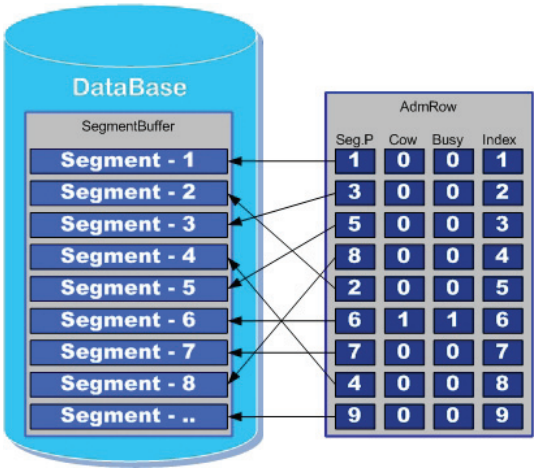


Figure 5.9: System architecture

Here the structure of the database is shown. At the highest level is the *AdmRow* table. This table contains information about segment pointers, Copy on Write flag and busy flag. The segment pointer is a number which tells the *BufferManager* which segment in the memory the index is pointing to. The reason this pointer is introduced, is because Copy on Write is used. When the system starts the *BufferManager* allocates several *Segment* objects, which are put in the *segmentBuffer* table. These objects create several direct buffers in the memory, one buffer for each segment. The total buffer size for this is the database size specified in the properties file, plus one extra segment which is used in Copy on Write. Code 5.3 shows the code for the creation of the *segmentBuffer* and the *AdmRow* table.

```
segmentBuffer=new Segment[numberOfSegmentes+1];
admRow=new AdmRow[numberOfSegmentes];

//Allocates the databasememory
while(i<numberOfSegmentes)
{
   segmentBuffer[i] = new Segment(this.segmentSize, loggerObj,i);
   admRow[i]=new AdmRow(i);
   i++;
}

//Allocate the free Segment memory
segmentBuffer[i] = new Segment(this.segmentSize, loggerObj,i);

freeSegment=i;
```

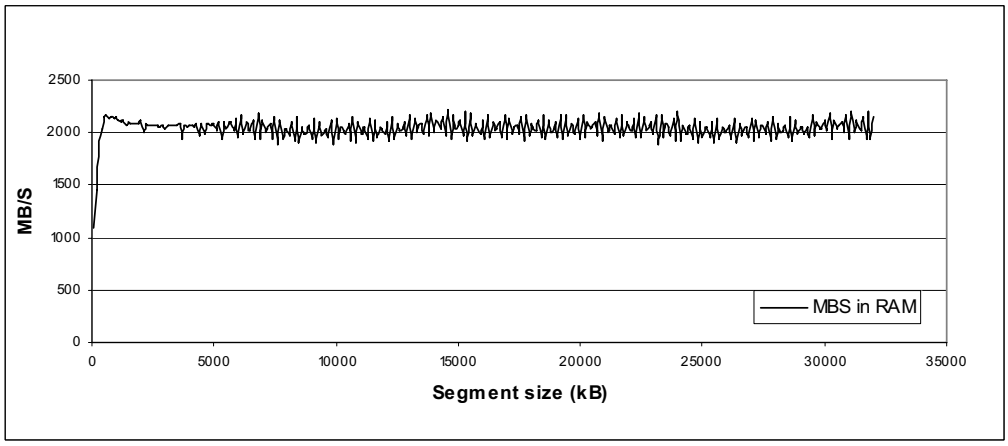
Code 5.3: Creation of segmentBuffer and AdmRow

In this system the checkpointer only locks one segment at a time. Since RDMA is used for transferring data, it is not possible to use several checkpointing threads. When using RDMA, memory space in the destination node is mapped from the source node. This technique does not allow for the possibility of using parallel RDMA threads. During a checkpoint the *Checkpoint* thread sets the "*Cow*" flag for the processed segment. If a transaction tries to access this segment at this time, it has to perform Copy on Write before it continues. Copy on Write copies the content of the flagged segment over to the separate free space allocated at the startup of the system. If the system has multiple threads that run transactions, only one can perform Copy on Write.

All the other threads have to wait until the segment is copied and this is done by using the "*busy*" flag. When a transaction is going to perform Copy on Write it marks the segment by setting the "*busy"* flag to true. If other transactions are going to access the same segment, they first check the *"busy"* flag and wait until the flag is set to false. To determine how long they must wait before they check the flag again, they use the Equation 5.4. A Copy on Write operation is done with approximately 2 GB/s. *SegmentSize* is given in bytes and *TimeToWait* in milliseconds.

$$TimeToWait = \frac{SegmentSize}{1024^2 * 2} \qquad (5.4)$$

To validate that this formula is true, the actual Copy on Write time has been compared to values from this formula. This validation has shown that this value is very well adjusted to the actual Copy on Write time. This is due to the transfer rate being very stable in the main memory (Graph 5.2).



Graph 5.2: Throughput measure of Copy on Write

After Copy on Write, the segment pointer in *AdmRow* is updated to point to the new segment. Next time Copy on Write is activated this segment is used as destination for the Copy on Write operation. When client applications are operating on the database they only communicate with the *BufferManager* class. For example if an application wants to update the database it only uses the command:
buffermanagerObj.writeToSegment(Segment,position,byte,Thread);
then the *BufferManager* takes care of the rest (Code 5.4).

```
public void writeToSegment(int segment,int position, byte value, ClientThread
CThread )
  {
       while (admRow[segment].getBusy())
       {
        try
        {
           loggerObj.logMessage(CThread.getName() + " is sleeping");
           CThread.sleep(((segmentSize) / (1024 * 1024)) / 2);
        }
        catch (InterruptedException e)
        {
           loggerObj.logMessage(e.getMessage());
        }
      }
      if(admRow[segment].getcow())
      {
          preformCopyOnWrite(segment);
      }
}
```

Code 5.4: writeToSegment()

The *Checkpoint* also uses the *BufferManager* when checkpointing the database. This thread uses the methods *lock()* and *unlock()* to manipulate the Copy on Write flag. These methods manipulate the *COW* parameter in the *AdmRow* class. To ensure consistent data in the database the *setCow()* method in the *AdmRow* class is synchronized. If use of another transfer technology, supporting multiple checkpointing threads, is introduced in the model, it is important to ensure that every thread gets the right information. This can be done because every segment has an *AdmRow* object containing information concerning the segment (Code 5.5). One of the most important parts of the system is implemented in the *BufferManager*. This is the Copy on Write algorithm. This algorithm will have great influence on the overall system performance.

```
private synchronized void preformCopyOnWrite(int admRowValue)
{
     admRow[admRowValue].setBuzzy(true);

     int avSegment=freeSegment;
     int lockedSegment=admRow[admRowValue].getsegmentPointer();
     admRow[admRowValue].setsegmentPointer(avSegment);
     admRow[admRowValue].setcow(false);

     segmentBuffer[avSegment].setByteBuffer(
          segmentBuffer[lockedSegment].getByteBuffer());

     freeSegment=lockedSegment;
     COWcounter++;

     admRow[admRowValue].setBuzzy(false);
}
```

Code 5.5: Copy on Write algorithm

The *preformCopyOnWrite()* method is automatically called when needed by the *BufferManager*. The algorithm itself is quite simple. First a lot of parameters are defined. In this system parallel checkpointers are not used and that is why the *freeSegment* can be set in the Copy on Write algorithm. The *freeSegment* is set to the segment locked by the checkpointer. But since there are no other checkpoints in the system this segment will not be needed until the checkpointer has unlocked it and locked another segment. By setting the *freeSegment* in this algorithm there is absolute control over the crucial update of this important variable. When this is done, the copying starts by using the segment commands *setByteBuffer()* and *getByteBuffer()* in the *Segment* object. Since this algorithm copies one byte at a time the prediction is that the system load will increase when the segment size increases. This will implicate more bytes to copy each time. With a constant database size the probability of hitting a flagged segment will also increase, together with the size of the segments.

5.1.3 ClientThread and TransactionController

ClientThread is the class that simulates the transaction load on the database. To be able to simulate different kinds of loads on the system, each client is simulated by a thread. The number of clients connected to the system is decided in the properties file on start up. The *DataBaseControl* reads the properties file and then creates the *TransactionController* object. This is the class which controls the client threads. The method head for the constructor in this object can be seen in the Code 5.6.

```
public TransactionController(int DBSizeKB,
                             int SegSizeKB,
                             int numberOfThreads,
                             int numberOfOperationsPerThread,
                             BufferManager bufferManager,
                             DataBaseControl dbControl,
                             DBMSLogger loggerObj,
                             int load)
                      throws java.lang.Exception
```

Code 5.6: TransactionController constructor

Here the number of parallel threads (clients) and how many operations each client is supposed to perform and other variables are defined. The variable load comes from the properties file and is read by the *DataBaseControl* when the *TransactionController* object is created. This parameter tells whether uniform or distorted transaction load is going to be used. The load parameter is forwarded to all client threads. In this version of the software the client threads will not perform any read operations. This is because such operations have no impact on the performance. Therefore all the operations are updating the database. When the *TransactionController* thread is started it creates *ClientThread* objects. The constructor of each client thread consists of a reference to the *DataBaseControl* object, the *BufferManager* object and the *DBMSLogger* object (Code 5.7). In addition, there are some other parameters used for administration of the threads.

```
public ClientThread(DataBaseControl databaseControl,
                    BufferManager buffermanager,
                    DBMSLogger logger,
                    int threadNumber,
                    int numberOfThreads,
                    int segSizeKB,
                    int numberSegments,
                    int load,
                    int rounds)
```

Code 5.7: ClientThread constructor

Each of the clients lives their own lives. When they are started they run until all of the defined operations are performed. The *TransactionController* waits for all the threads to finish before it finishes. This is done by using the command *Thread.join()*; this command waits until the thread has finished (Code 5.8).

To manage all the *ClientThreads* the *TransactionController* has an array which consists of all the thread objects.

```
        while(cont)
        {
            try
            {
                //wait until the thread is finished
                threadArray[counter].join();
                if (counter == 0)
                    cont = false;
                else
                    counter--;
            }
            catch(java.lang.InterruptedException e)
            {
                System.out.println(e.toString());
            }
       }
```

Code 5.8: TransactionController waiting for clients to finish

When the first thread is finished it goes on and checks if thread number two is done working. This loop continues until all the threads are finished. The waiting itself is activated by using the predefined Java command *Thread.sleep()*. Since each *ClientThread* has its own timer which is stopped when the thread finishes, it has no impact on the final timing results. The only job for the client is to update a random byte in a random segment. To make it a more real transaction load, each client is set to update the database 500 times for each write operation. The only possible problem of updating a segment is if the chosen segment is locked by the checkpointer. If this is the case, a Copy on Write operation will be performed to avoid blocking of the client. If this happens the client will have to wait until the segment is copied before it continues.

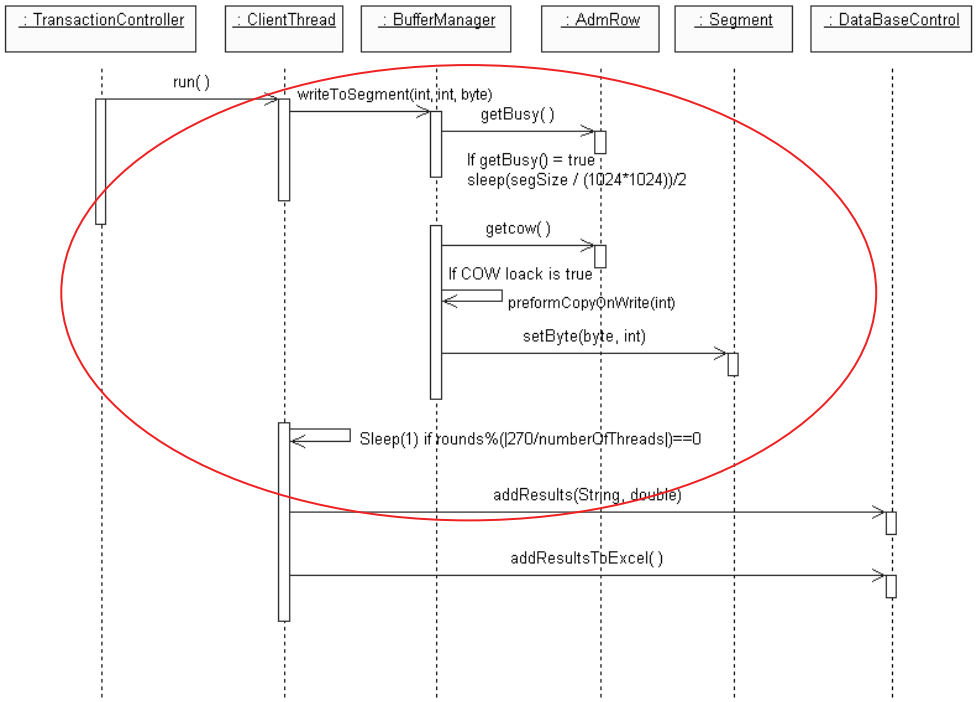


Figure 5.10: Sequence diagram showing ClientThread

As can be seen from the sequence diagram (Figure 5.10); if the segment is not locked by the checkpointer then the client will update the segment and then sleep according to Equation 5.5. The sleep function is to simulate some of the overhead found in a real network. This loop (see red ring) is performed as many times as specified in the properties file. This is the way every *ClientThread* works. All the different clients will perform in parallel, giving the situation illustrated in Figure 5.11.

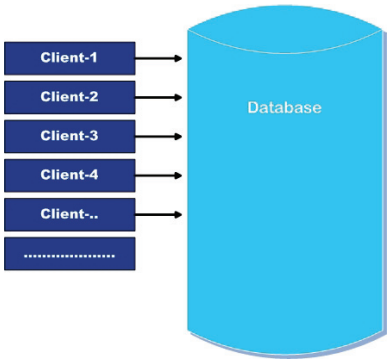


Figure 5.11: ClientThreads working on the database

To find random bytes and segments the Java *Math.random()* function is used. Since all of the values have different limits the function has to be run three times, one for each random value. Two of these operations can be seen below in Code 5.9.

```
wichByte=(int)(Math.random()*10);
wichSegmentByte=(int)(Math.random()*segmentSize);
```

Code 5.9: WichByte and wichSegmentByte function

In some cases it can be useful to simulate distorted load on the system. This is done by changing the *loadType* parameter which determines which of these two functions to run.

```
if(loadType==0)//Uniform load
   whichSegment=(int)(Math.random()*(numberOfSegments));
else //Disorted load
   whichSegment=(int)(Math.random()*(numberOfSegments)*
   ((threadNumber+1)/numberOfThreads));
```

Code 5.10: WhichSegment functions

To give a better understanding an example will be given. In the example there is a system with 10 segments and 5 threads which is set to have a distorted load.

Thread	Number of segments to choose from
Thread 0	10 * (1/5) = 2
Thread 1	10 * (2/5) = 4
Thread 2	10 * (3/5) = 6
Thread 3	10 * (4/5) = 8
Thread 4	10 * (5/5) = 10

Table 5.1: Distorted load

As can be seen in the table above the data will be divided distortedly. This helps to give a better simulation of a real database system. Which byte to put into the database is decided by choosing a number between 1 and 10. This number corresponds to the index in an array which consists of 10 different bytes. When this is done the thread writes to the segment. This is done by calling the method *writeToSegment()* in the *BufferManager*. The *BufferManager* checks if the segment is locked by the checkpointer. If so, Copy on Write is performed. Then the method *setByte()* method in segment is called and the database is updated.

To navigate through all of the segments the *BufferManager* looks in the *AdmRow* class. This class manages the table which refers to pointer to the corresponding segments. The reason this table is needed is because these pointers change when Copy on Write is performed.

Start of program

AdmRowIndex	SegmentPointer
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

End of program

AdmRowIndex	SegmentPointer
1	9
2	8
3	10
4	4
5	5
6	11
7	7
8	2
9	1
10	6

Table 5.2: The effect of Copy on Write

As can be seen from Table 5.2, these pointers can be moved around in a very strange order by the time the program has finished. If the *ClientThreads* are performed in back to back mode, running without any "sleep" to slow them down, the CPU usage will lie somewhere between 97 % and 100 %. This is not acceptable since such stress of the system causes other uncontrolled factors to play a role on the database performance. To change this situation a *sleep()* function is induced.

$$DecideSleep = r\%\left(\frac{Z}{numberOfThreads}\right) \qquad (5.5)$$

$$r = TotalNumberOfRoundToRun - finishedRounds$$

Z is a constant defined to be 270. It determines how often the *ClientThread* will sleep. Lower values of *Z* will give a higher occurrence of sleep. If the *DecideSleep* value equals zero the *ClientThread* sleeps for one millisecond. This formula was developed by several runs and adjustments. The goal was to create a scalable function concerning the number of threads. It is difficult to decide what the desirable number of threads should be. Due to the scalability of this function, the CPU usage will be approximately the same, despite the number of threads.

Results show that this function gives stable results in different situations. The CPU usage is lowered to around 60 – 70 % which is found to be acceptable. This is done to avoid the JVM to be an important factor in the measurements.

The *ClientThread* is timed on each operation it performs and these results are used to calculate the TPS. The TPS is one of the parameters that have their confidence interval calculated. To make the calculation job easier the TPS for each operation is calculated continuously.

$$TPS = \frac{1}{timeToPerformOneOperation} \qquad (5.6)$$

The *timeToPerformOneOperation* parameter is calculated in seconds. This TPS value is added up continuously as the systems runs, as is the TPS^2 value. These sum values are used to calculate the standard deviation (done by Equation 5.1). When the client has finished it stores the results in the result parameter in *DataBaseControl*. This result contains the time the client has spent on performing all its operations. *ClientThread* communicates with four classes, *DataBaseControl*, *BufferManager*, *HiResTimer* and *Logger*. *ClientThread* objects are created and started by the *TransactionController* class. A graphical summary of the life of a *ClientThread* can be found in Figure 5.12.

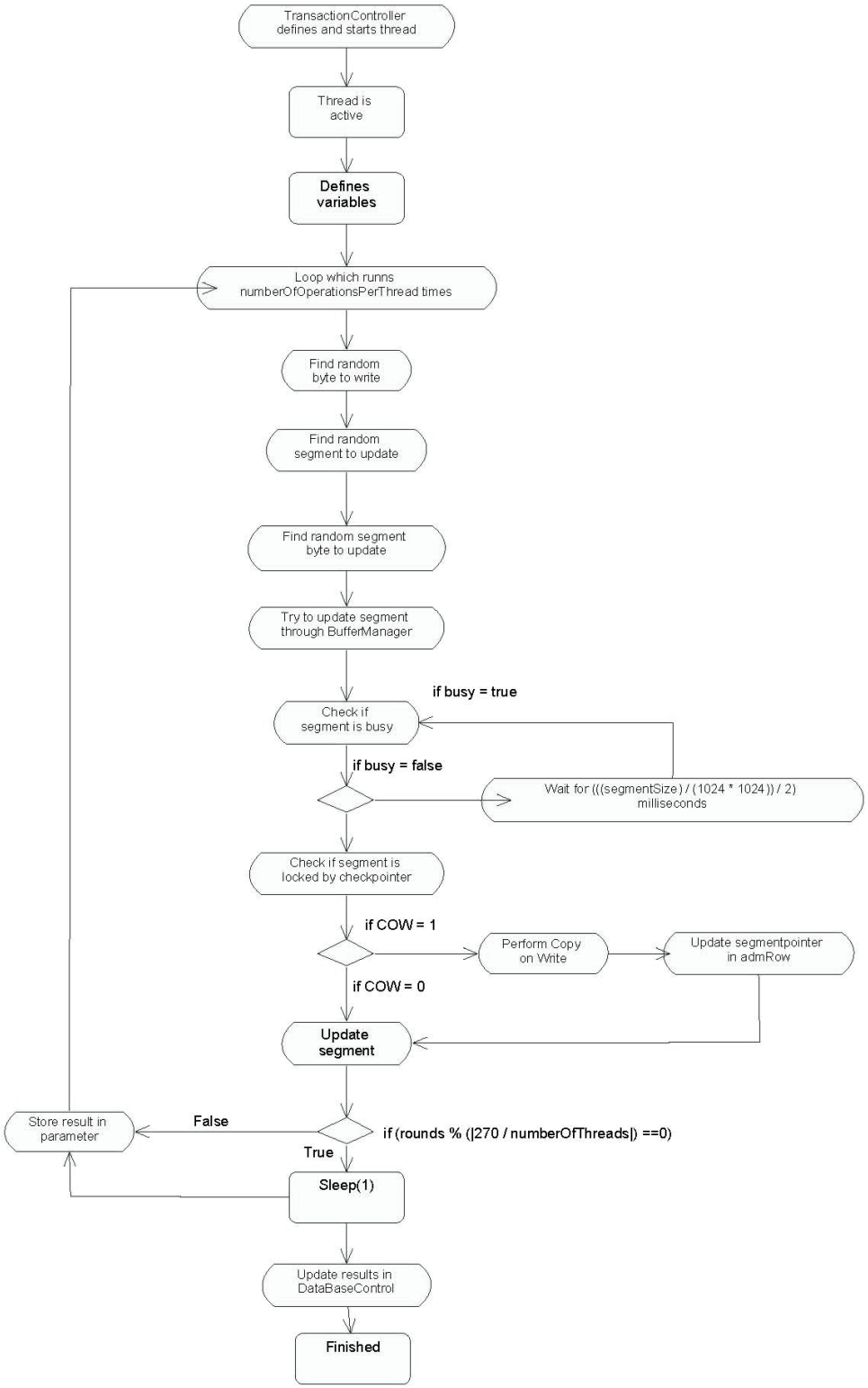


Figure 5.12: Activity diagram ClientThread

5.1.4 Checkpoint

The *Checkpoint* class frequently performs a checkpoint. After a checkpoint is performed, the *Checkpoint* class sleeps for a time, given by the properties file, before it performs a new checkpoint. This class is initialized by the *DataBaseControl* class and run as a separate thread. The constructor for the *Checkpoint* object can be seen in the code below.

```
public Checkpoint(BufferManager bufferManagerObj,
                      Properties properties,
                      DBMSLogger loggerObj,
                      int segSize,
                      int DBSize,
                      DataBaseControl dbObj)
```

Code 5.11: Checkpoint constructor

When an object of the *Checkpoint* class is made, the constructor saves a reference to the *BufferManager* object. This object contains the database that the checkpoint is performed on. The *Properties* object contains configuration information. The *DataBaseControl* executes the *Checkpoint* thread by running the *run()* method. This method runs an infinite loop and sequentially sends the segments to the *IOController.* The system concerning the checkpointer is made very flexible, meaning that the data transfer part of the checkpointer is very easy to replace by something more feasible than RDMA, e.g. disk.

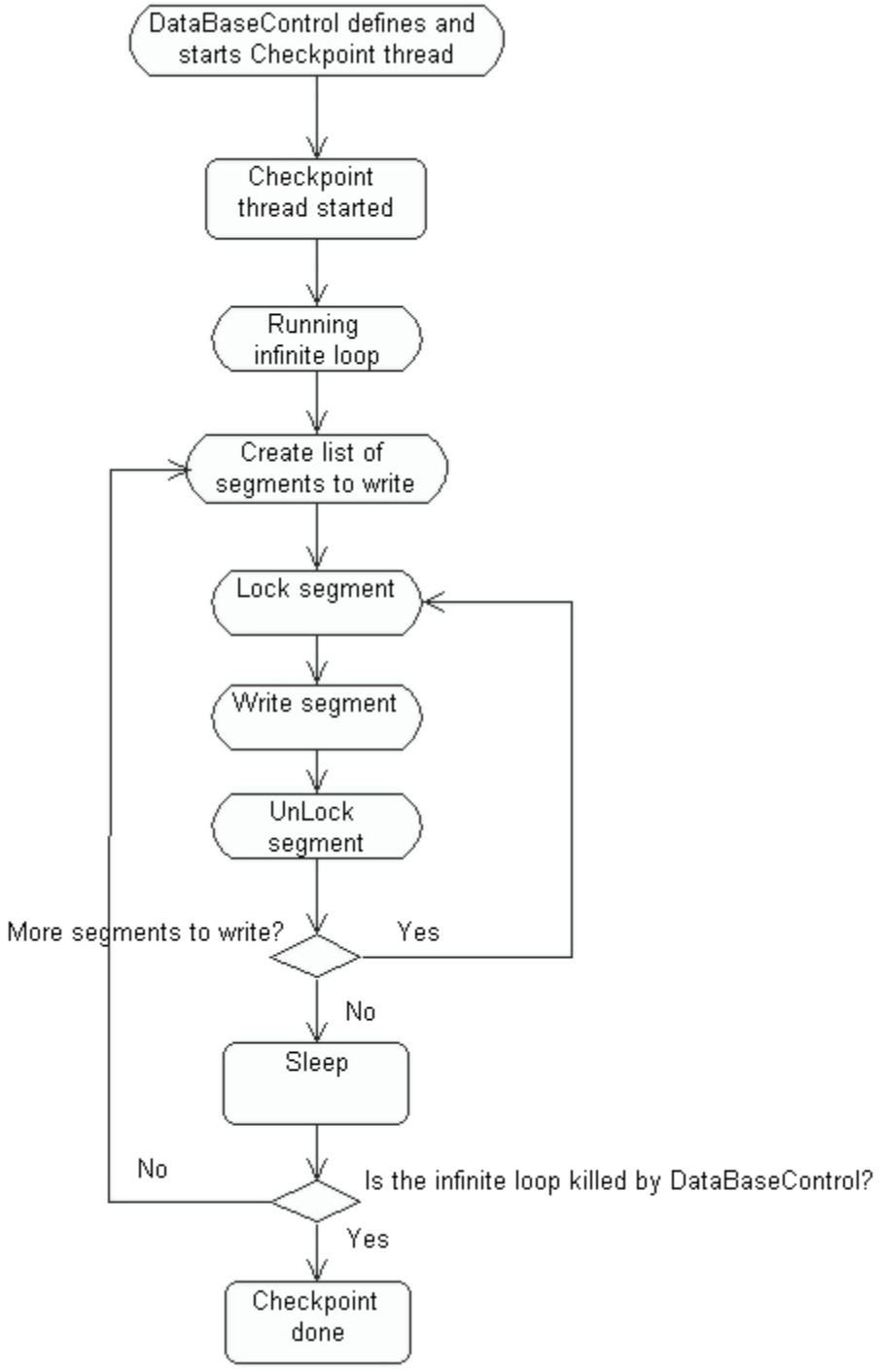


Figure 5.13 Activity diagram for checkpointer

Before it sends the segment to the *IOController*, it sets a Copy on Write flag for the segment in the *BufferManager* (Figure 5.13). This flag notifies the *BufferManager* when transactions try to write to the segment during an *IOController* process. When the *IOController* is done with the segment, it unsets the Copy on Write flag again. After a lot of testing, it has been decided to run the checkpointer back-to-back.

This means that when a checkpoint has finished it just starts directly on another one. This is done because tests show that this is what gives the best results. When the checkpoint sends the segment to the *IOController*, it waits for the *IOController* to finish, before it proceeds. In Figure 5.14 the sequence diagram for the checkpoint can be seen.

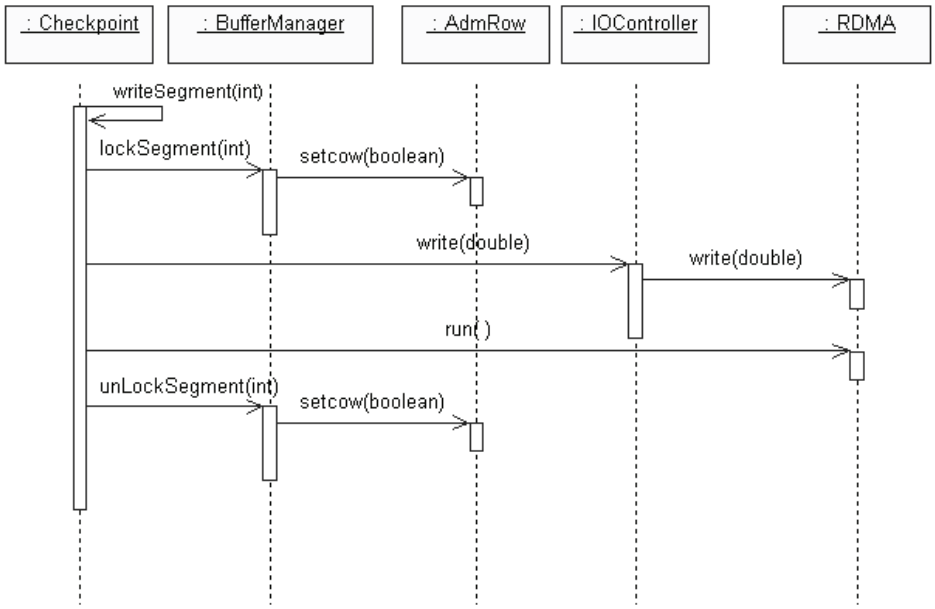


Figure 5.14: Sequence diagram for checkpointer

When the checkpointer finishes it stores the timing results in *DataBaseControl*.

5.1.5 IOController and RDMA

IOController is a common interface for *RDMA* and other IO methods that the model should support. This model can be extended to support writing to disk and other media, by adding new sub classes. In this implementation the model only supports RDMA. The IO methods run as a separate thread. There are two common methods for all IO controllers. First is the *write()* method that saves information about the data that will be processed. This can be pointers or values, depending on the implemented subclass. This program has implemented a simulation of the RDMA protocol.

The *RDMA* class simulates the time spent by a real RDMA protocol. This is done in two steps. First it saves how large the transferred segment is and calculates the transfer time. Then it is executed by the *Checkpoint* class and waits the calculated time, before the thread ends.

RDMA uses 500 µs to set up the connection and have a bandwidth at 1 GB/s. To calculate how long it takes to transfer a segment it uses this equation:

$$time = 0.5 + \frac{size * 1000}{1024^2} \qquad (5.7)$$

"*size*" is the size of the segment in kilo byte. The first constant in the equation is the time for establishing the connection. 0.5 means 0.5 milliseconds or 500 µs. The real RDMA accesses the memory directly and reduces the CPU load related to protocol processing. For this reason it is good enough to use the *Thread.sleep(time)* method to simulate transfer. This method does not use the CPU and waits the given time before it continues.

5.1.6 CPUusage

To calculate the CPU usage the program uses Java code by Vladimir Roubtsov. This code uses C code to get CPU clock information and Java Native Interface to connect to the C program from a Java program [Rou02]. The *SystemInformation* class declares a native method, which returns the number of milliseconds of CPU time used by the current process so far (Code 5.12).

```
public static native long getProcessCPUTime ();
```

Code 5.12: getProcessCPUTime()

This native method uses the *GetProcessTimes()* and adds CPU time spent executing kernel and user code on behalf of the current process, normalizes it by the number of processors, and converts the result to milliseconds. The equation for this calculation is:

$$ProcessCPUTime = \frac{KernelTime + UserTimer}{NumberOf\ \mathrm{Pr}ocessors * 1000} \qquad (5.8)$$

Note that the *ProcessCPUTime* is the CPU time used since the creation of the JVM process. By itself this data is not particularly useful for profiling.

To get the CPU usage the program uses Java methods to take snapshots at various times and report CPU usage between any two time points. The system time since last snapshot divided on the CPU usage time since last snapshot, shows how much of the system time that is spent on the current thread.

$$ProcessCPUUsage = \frac{\Delta CPUTime}{\Delta SystemTime} \qquad (5.9)$$

When JVM loads the native code, it initializes the global variables s_*currentProcess* and *s_numberOfProcessors*. This is done in a method referred to as *JNI_OnLoad()* and is shown in the code below.

```
static HANDLE s_currentProcess;
static int s_numberOfProcessors;

JNIEXPORT jint JNICALL
JNI_OnLoad (JavaVM * vm, void * reserved)
{
    SYSTEM_INFO systemInfo;

    s_currentProcess = GetCurrentProcess ();

    GetSystemInfo (& systemInfo);
    s_numberOfProcessors = systemInfo.dwNumberOfProcessors;

    return JNI_VERSION_1_2;
}
```

Code 5.13: JNI_OnLoad()

The C program is packed into a native library (silib.dll) and the *SystemInformation* class connects to it by Code 5.14.

```
private static final String SILIB = "silib";

static
{
   try
   {
     System.loadLibrary (SILIB);
   }
   catch (UnsatisfiedLinkError e)
   {
     System.out.println ("native lib '" + SILIB + "' not found in
     java.library.path': "+ System.getProperty ("java.library.path"));
     throw e; // re-throw
   }
}
```

Code 5.14: loadLibrary()

The CPU usage information is saved in the *accept()* Java method. This method is implemented by the user. In this case, the CPU monitor is executed by *DataBaseControl* and the *accept()* method in this class records the CPU usage and calculates the mean value of all the samples.

5.1.7 HiResTimer

To perform high resolution timing the program uses Java code by Kevin T. Manley. One of the critical aspects with the DBMS is the timing. In order to get good results it is important to have the ability to time each operation with a high resolution timer [Man01]. This system runs on Windows XP. Java has a timer function called *currentTimeMillis()*, but this function uses the operating system function *GetTickCount()*. The resolution on this timer will therefore change according to different operating systems. To determine the resolution when using Windows XP a test program was used (Code 5.15).

```
import java.io.*;

public class TimerTest
{
        public static void main(String[] args)
        {
                long time = System.currentTimeMillis();
                long time2 = 0;
                while (time2 == 0)
                {
                        time2 = System.currentTimeMillis()-time;
                }
                System.out.println("granularity of system timer is " + time2
                + " milliseconds");
        }
}
```

Code 5.15: TimerTest class

This program records the time by using the function *currentTimeMillis()* then it runs a *while loop* until the difference between this value and the next timer value exceeds zero. Running this program under Windows XP gives a time slice of 15 - 16 milliseconds. In most cases this will be an applicable timer, but in this system it will be too coarse. In the simulation of the RDMA an establishing time that is 500 microseconds, or 0.5 milliseconds will be used. It will therefore be desirable to have a higher resolution timer.

To be able to create a high resolution timer Java Native Interface (JNI) need to be used. JNI makes it possible for Java code to call functions in C++ programs. The high resolution timer was implemented in C++ and then the JNI was used to access these functions. There are two functions needed to implement this timer, *QueryPerformanceFrequency* and *QueryPerformanceCounter*. *QueryPerformanceFrequency* returns the frequency of the high resolution counter in cycles per second. *QueryPerformanceCounter* retrieves the current counter value. The high resolution timer is written in C++ built into a WIN32 DLL that wraps the *QueryPerformanceFrequency* and *QueryPerformanceCounter* functions. The C++ code consists of three important parts. One function is to determine whether the high resolution timer exists or not. If it does not exists the system will go back to use the *GetTickCount()* and function in the same way as the *currentTimeMillis()*. There is one function to start the timer and one function to stop the timer. These native method heads are then implemented in the *HiResTimer* Java file (Code 5.16).

C++ code

```
JNIEXPORT jboolean JNICALL    Java_HiResTimer_isHighResTimerAvailable
(JNIEnv *, jobject)

JNIEXPORT jdouble JNICALL Java_HiResTimer_startTiming  (JNIEnv     *,
jobject)

JNIEXPORT   jdouble   JNICALL   Java_HiResTimer_endTiming   (JNIEnv   *,
jobject, jdouble dStart )
```

Java code

```
public native boolean isHighResTimerAvailable();

public native double startTiming();

public native double endTiming( double dStart );
```

Code 5.16: C++ code and corresponding java code

When this is done the native methods are called in the same way any other Java method is called. In the implementation of the *HiResTimer* two additional methods are added, *startTimer()* and *endTimer()*. These functions use the native methods and make the use of the timer more perspicuous. The program first calls the *startTimer()* function then the *endTimer()* function. When calling the *endTimer()* function the difference in time between *startTimer()* and *endTimer()* is returned as a double value.

In the DBMS several timers are created, because each object has it own timer. The implementation of the C++ code is quite simple (Code 5.17 and Code 5.18).

```
JNIEXPORT jdouble JNICALL Java_HiResTimer_startTiming
(JNIEnv *, jobject)
{
        LARGE_INTEGER li;
        if( HiResCounter.Exists() )
        {
                QueryPerformanceCounter( &li );
                return (double)(li.QuadPart);
        }
        else
        {
                return (double)GetTickCount();
        }
}
```

Code 5.17: Java_HiResTimer_startTiming

```
JNIEXPORT jdouble JNICALL Java_HiResTimer_endTiming
  (JNIEnv *, jobject, jdouble dStart )
{
        LARGE_INTEGER li;
        if( HiResCounter.Exists() )
        {
                QueryPerformanceCounter( &li );
                return ((li.QuadPart - dStart) * 1000.0 /
                HiResCounter.Freq());
        }
        else
        {
                return (GetTickCount() - (DWORD) dStart);
        }
}
```

Code 5.18: Java_HiResTimer_endTiming

The *startTiming()* returns a *jdouble* which is the current counter value. This return value is later used as a parameter in the *endTiming()* function (called *dStart*). The *endTiming()* function then returns the difference in time between the *dStart* value and the *QueryPerformanceCounter()* value formatted to the system. From the Java code the method will return the time in milliseconds with a high resolution.

5.2 Result management and graphs

After the program has completed a run, it saves a file prepared for import in Microsoft Excel. This file is imported into a sheet in Ms Excel on a different computer over the network. This import is performed periodically to get the newest data. By using another computer on the network to present and process the result, the computer running the simulation uses a minimum of CPU time on tasks related to presentation. The data file itself is contains nothing else then numbers collected from the simulation. A sample of one this type of file can be found in Appendix III. This sample show the total output after a run with five threads and uniform load and justify the need to represent the results in graphs in stead of raw data.

The Ms Excel sheet contains graphs to plot the data after is has been imported. These graphs are the same for all the runs and show collected information about CPU usage, TPS and checkpoints. To update these graphs the user can run a macro after each import. This macro updates columns containing calculations and graphs.

Some data is calculated from the imported ones. For example, MB/s that is transferred during a checkpoint is calculated by dividing the database size (1 GB) by the checkpoint time (imported). Other columns that are calculated are:

MB/s in RDMA	$\frac{DBsize}{CheckpointTime}$
MB/s in RAM	$\frac{SegmentSize}{COWTime}$
Theoretical COW time	$\frac{SegmentSize/1024}{2}$
Checkpoint time upper confidence interval	$Checkpointtime + ConfidenceInterval$
Checkpoint time lower confidence interval	$CheckpointTime - ConfidenceInterval$

COW time lower confidence interval	*CowTime – ConfidenceInterval*
COW time upper confidence interval	*CowTime + ConfidenceInterval*

The cells containing these formulas are updated by the macro. When new data has been imported, the macro runs through every row for these columns and inserts the correct formula. Code 5.19 shows the pseudo code for how the cells are updated with the right formula.

```
From FirstRow To NumberofImportedRows Step 1
        If ("Imported Row has a value") Then
                Insert the formulas for every column.
        End If
Next
```

Code 5.19: Pseudo code for updating cells in the sheet

After updating all the cells, the graphs are updated. The graphs can vary in number of data series and name of the graph. For each graph the data range is redefined for each data series to cover all the current data values. The pseudo code for updating a graph is in Code 5.20.

```
SelectChartObject("Name of chart")
UpdateXandYValueForSerie(1)
UpdateXandYValueForSerie(2)
UpdateXandYValueForSerie(..)
UpdateXandYValueForSerie(n)
```

Code 5.20: Pseudo code for updating charts, where n is number of series in the chart

The charts that are used are TPS, COW time and checkpoint time with confidences interval, CPU usage, number of COW and checkpoints and MB/s in RAM and RDMA. Some of these charts are used to validate the model, while others are used to conclude this report.

5.3 Validation

When developing software the most important phase is test. The test phase is explained in detail in Appendix I. When developing a simulator the test phase plays an even more important role. To be able to intemperate the results from the simulation it is important to be 100 % sure that the results are accurate. This thesis has tried to minimize every source of error in several different ways.

5.3.1 Detection of possible pitfalls

During the design phase all possible pitfalls (Appendix I) were searched for and detected. Those detected were analyzed. When the analyzing phase was complete, suggestions to solutions were presented. Then, all of these possible problems were prototyped to verify the solution suggested. This resulted in a solution to all of the detected possible pitfalls. By doing this the quality of the system was improved and the chance of being delayed because of pitfalls was minimized (*HiResTimer* and *CPUUsage* are examples of modules tested before implementation).

5.3.2 Manual inspection of code

Static testing was also performed in this thesis. This static testing was performed by inspections of the source code. The inspections were performed by both developers to ensure a common understanding, to detect errors and find possible solutions to optimize the code.

5.3.3 Oral explanation of the code

A good way to ensure that all the developers have a common understanding of how the code works is to give each other an oral explanation of what happens. Simultaneous with this oral presentation it is possible to check if the documentation of the code contains enough information. In this way both the quality of the product and the documentation is ensured.

5.3.4 Inspection of the debug log (white box testing)

The most comprehensive testing in this thesis was done using white box testing. This is a kind of dynamic testing. In this test the debug log is activated. This function logs every action in the system. After one run there are several MB of log to inspect. The debug log gives the developers the possibility to follow every action inside the program and in this way ensure that everything is happening in the correct order and in a correct way. The *admRow[]* and the Copy on Write algorithm was validated by inspecting the debug log. The debug log was used to ensure that all the pointers were in the right place after a Copy on Write operation. An extract of a debug log for a test with five threads can be found in Appendix IV. This extraction shows the start of the program and a beginning of a checkpoint (first seven segments). It shows the complexity and extent of the log after only seven of 100 segments have been processed during the first checkpoint.

5.3.5 Black box testing

The opposite of white box testing is black box testing. In this test only the output and input data are studied. One way of doing this is to run several tests with the same parameters. In that way the input and the expected output are known. By doing so, the stability of the system will be tested. Another possibility is to know the input and have an idea about what the output should be. This is done by running boarder cases, e.g. one thread, and ensures that the system works satisfactorily.

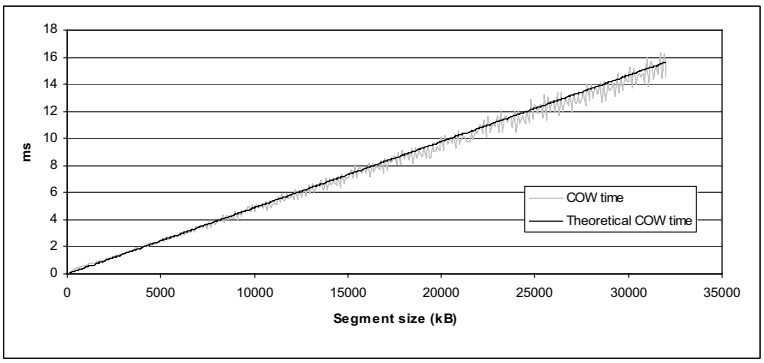
5.3.6 Regression testing

During the whole thesis, regression testing was performed to ensure that new components in the system did not come in conflict with existing components.

5.3.7 Graphical validation

When developing the Copy on Write mechanism it was detected early on that a blocking had to be made for other transactions when the segment was copied internally in the memory. To ensure that client threads do not wait longer than necessary, a function for calculating this waiting time was developed.

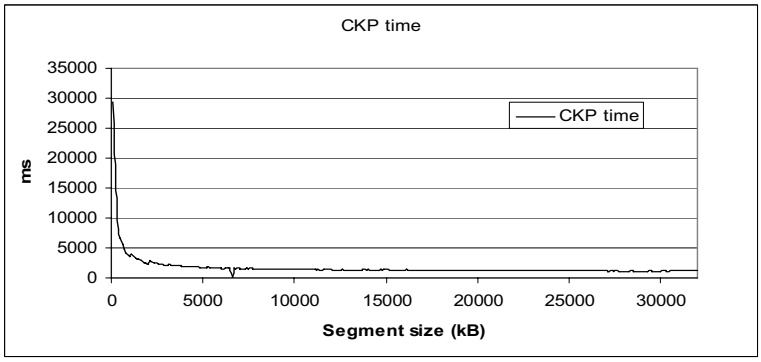
This function is further explained in Chapter 5.1.2, Equation 5.4. This formula is based on a transfer rate of 2 GB/s internally in memory. To validate this formula, this value was plotted together with the actual Copy on Write time on a graph. An example of such a graph can be seen underneath in Graph 5.3.



Graph 5.3: Theoretical COW and measured COW

From this graph the actual Copy on Write time can be seen as the grey line while the theoretical time can be seen as the black line. It is not difficult to see that the formula gives a very good estimation of the waiting time.

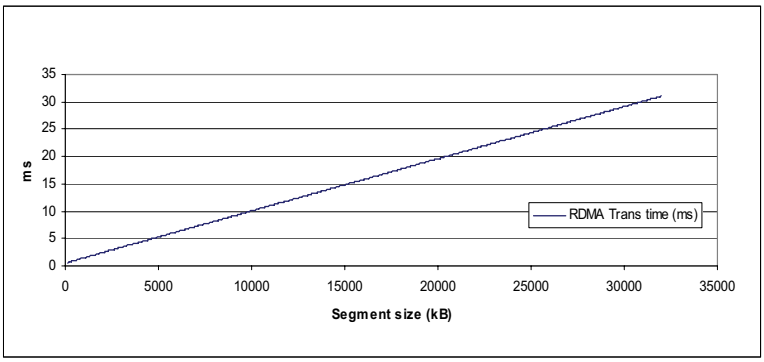
An important part of this thesis is the simulation of RDMA over InfiniBand. For this simulation a transfer rate of 1 GB/s is chosen. This means that a checkpoint time of around 1000 ms will be the ideal result, since all tests will be run with a database of 1 GB. To validate this parameter the actual checkpoint time is plotted in a graph like Graph 5.4.



Graph 5.4: Checkpoint time

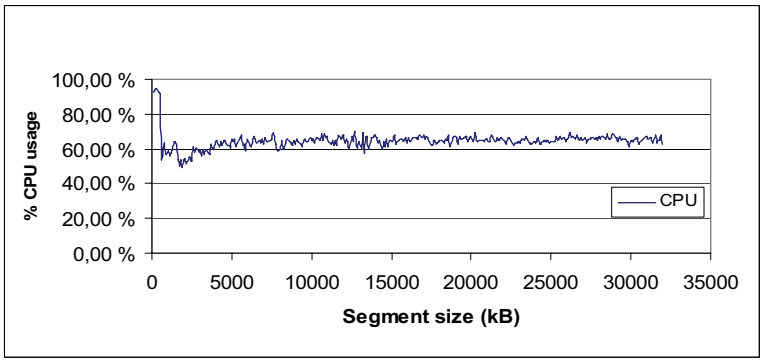
As can be seen from this graph, the checkpoint time is very high in the beginning but stabilizes at around 1000 ms. This graph shows that the RDMA function is working and gives the results that were expected.

Another way the RDMA function has been validated is the total transfer time per segment. This is a calculated value that should give a continuity rising graph. The value is calculated from Equation 5.7 and was plotted in Graph 5.5. It shows a continuously rising graph as predicted.



Graph 5.5: RDMA

From this graph it was possible for the developers to conclude that they had made a satisfying RDMA function. Another difficult part of this system has been to simulate a realistic transaction load. One of the problems was to find a reasonable time to wait between every update operation for the client threads. If the system ran back-to-back it resulted in a CPU usage of 100 %. Such a high CPU usage makes the system loose control over the scheduling. The OS will automatically make some decisions the DBMS cannot control. To reduce the CPU usage Equation 5.5 was introduced. This formula is explained in more detail in Chapter 5.1.3. To validate this formula the results from each run were printed into a graph, such as Graph 5.6. With this graph it was possible to conclude that the formula gave a stable effect of reducing the CPU usage.



Graph 5.6: CPU usage

During the development of the formula Windows Task Manager was used to give a fast indication as to whether the function gave better results or not. In the task manager it is possible to see the total CPU usage on the system or to see the CPU usage for a chosen process (Figure 5.15).

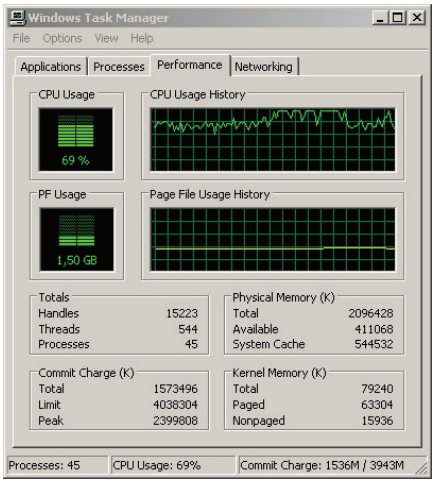


Figure 5.15: Task manager

Chapter 6

Results

The model represented in this master thesis has been run several times during the development. When the implementation was finished, several test cases were run with different parameters, to give a final result.

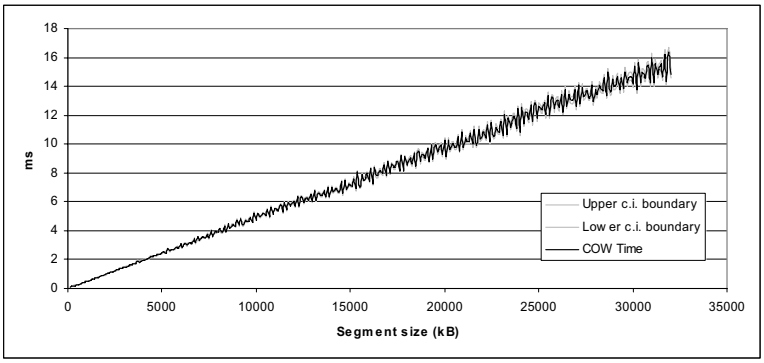
When analysing results from a simulation it is very important to get the hardware specification running the test cases. The specification helps analysing the results and to get a better understanding for the environment where the simulation has taken place. For this thesis the following hardware specifications were used during test.

Operating System	Windows XP Service Pack 2 (Build 2600)
Machine Type	AT/AT COMPATIBLE
System BIOS Version	A M I - 9000414
System BIOS Date	09/14/04
Processor Type	x86 Family 15 Model 3 Stepping 4 Intel Pentium 4, hyper threaded
Processor frequency	3.00 GHz
Processor Vendor	Genuine Intel
Number of Processors	1
Physical Memory	2048 MB
Java version	1.4.2_04

Number of threads and segment size are the parameters that have the biggest effect on the model. Therefore the following chapters give an analysis of the results from test cases with 1, 5 and 25 threads in parallel. To show the insignificant difference with distorted load, one test case has been run with five threads and distorted load. All the test cases are written with the same layout.

This is done because it gives an orderly way to compare all the results later in the discussion (Chapter 7).

There is one result that is common for all these test cases. The time it takes to perform Copy on Write in main memory has the same shape and value interval no matter how many client threads that run in parallel and what type of load they use. This result is represented in Graph 6.1 and shows that the time varies more the larger the segments are. The confidence interval (grey line) does not vary much, but the difference between neighbor values increases. The grey line is almost invisible because of the small confidence interval.



Graph 6.1: Time it takes to perform Copy on Write

The model can also use a different load on the database. To see if it makes any difference to the result, the last test case has been run with five threads and distorted load.

Before each run, the buffer is warmed up with one extra run with the first segment size. During black box testing, the difference in the first run showed that the model used very different time compared to neighbor values. The reason for this might be that the operating system took time to load the JVM.

In *clientThread* the sleep function simulates administrative costs and makes the CPU usage always stay well under 100 %. Because of the sleep function in *clientThread,* the CPU usage will stay stable at around 60 – 70 %. This is the case for the majority of the test cases because the sleep function is made scalable with the number of threads. The more threads that run in parallel, the more each one sleeps.

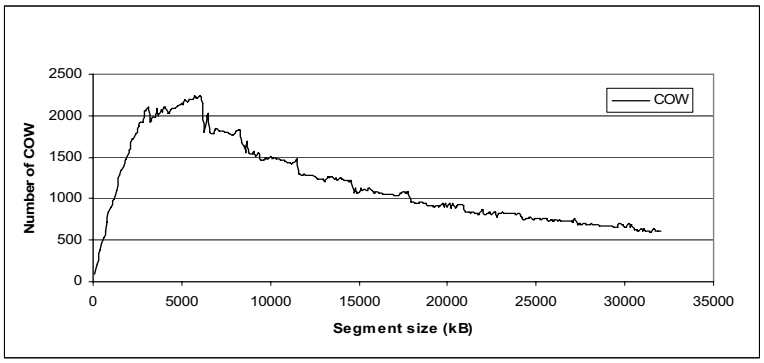
6.1 Uniform load with one thread

TPS per thread will have a higher value compared to runs with more threads. This is because with one thread there will be no competition for resources in the system. This means no waiting for other threads to finish or waiting for CPU time. A one threaded system is less stressed than a system with several threads. The common denominator for this test case is much variance with segments size lower than 5 MB.

6.1.1 Analysis of graphs

This test shows the results from the boundary case where there is only one thread. It has the shortest run-time and therefore the smallest attribute domain on checkpoints and Copy on Write operations.

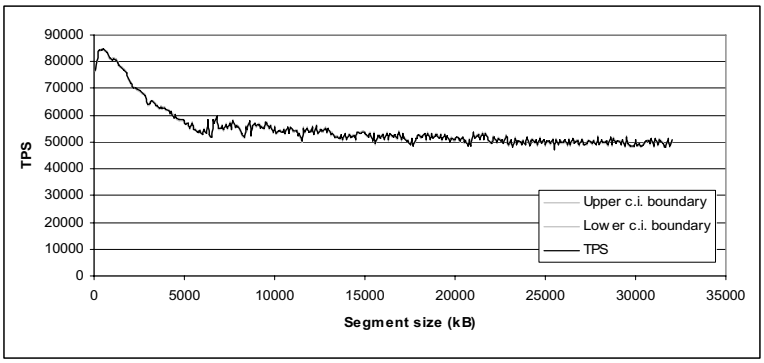
Copy on Write



Graph 6.2: Number of Copy on Write operations

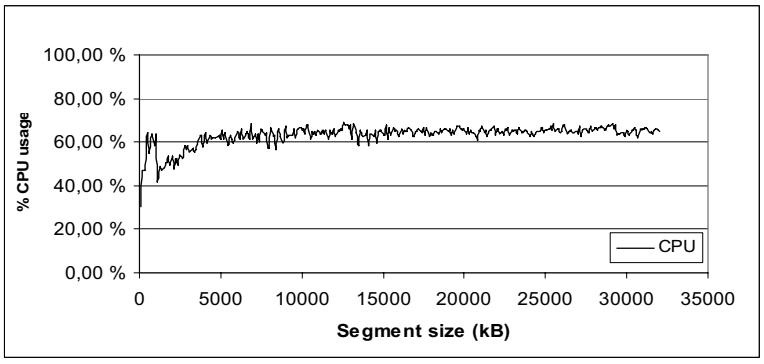
The total number of Copy on Write reaches a peak when the segment size is around 6 MB. At this time 2241 Copy on Write operations is performed. From this peak the graph falls continuously towards 600 Copy on Write operations. The graph shows a slight tendency to flatten out. The number of Copy on Write operations is between 95 and 2241 operations.

TPS and CPU usage



Graph 6.3: Transactions per Second

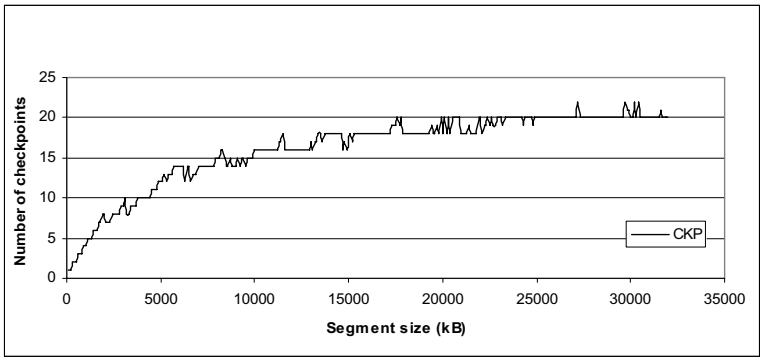
The TPS in Graph 6.3 has a peak at a segment size of 500 KB with 84677 TPS. From this point the graph decreases towards 50000 TPS which is reached when the segment size is around 10 MB. From this point and forward the graph flattens out. The TPS vary between 84677 and 47305, but have a small variation with an average confidence interval of 250. The CPU usage has some variation in the beginning. It increases from a low value towards 64 % and has a drop at 1100 KB with 41 %. This can also be seen as a local peak. From 5 MB it keeps stable at around 64 % usage (Graph 6.4).



Graph 6.4: CPU usage

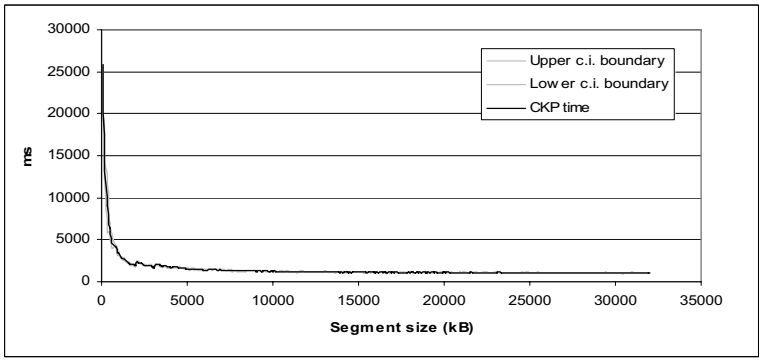
Checkpoint time and number of checkpoints

The number of checkpoints increases towards 20 at segment size of 23 MB. The graph shows a small amount of variance as regards its neighbor values (Graph 6.5).



Graph 6.5: Number of checkpoints

At about 10 MB, the checkpoint time flattens out at around 1000 ms or 1 second (Graph 6.6). The graph shows little variance in values with a mean confidence interval of 57.



Graph 6.6: Measured time to perform a checkpoint

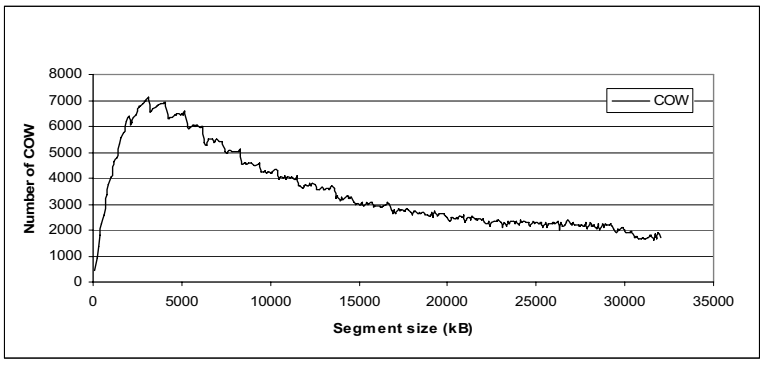
6.2 Uniform load with five threads

The total number of transactions will increase together with the number of threads. This implies that the number of checkpoints also will increase due to the increased run-time of the system. With an increasing number of threads it is likely to that there will be a bigger chance of triggering the Copy on Write mechanism. The results from this test case show stable results. A clear trend of where the system has stabilized is shown.

6.2.1 Analysis of graphs

The graphs have a tendency to show stable values after 10 MB. The most interesting trends and shapes lie between 100 KB and 10 MB.

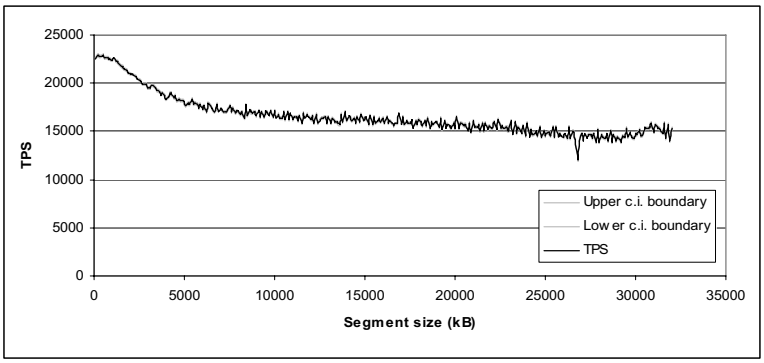
Copy on Write



Graph 6.7: Number of Copy on Write operations

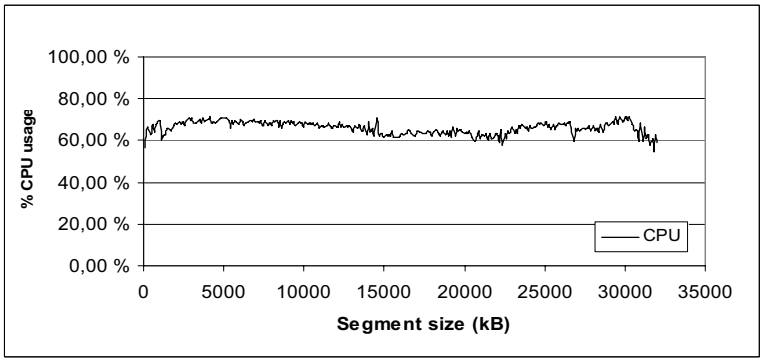
The total number of Copy on Write reaches a peak when the segment size is around 3 MB (Graph 6.7). At this segment size, up to 7100 Copy on Write operations are performed for each run. From this peak the graph falls continuously towards 2200 Copy on Write operations. The graph shows a slight tendency to flatten out after a segment size of 23 MB. The number of Copy on Write operations is between 1800 and 7100 operations.

TPS and CPU usage



Graph 6.8: Transactions per Second

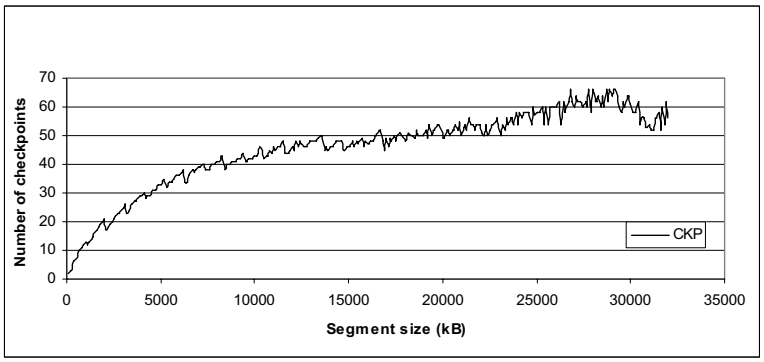
The TPS has little variation and decreases from 22500 towards 14000 at a segment size of 28 MB (Graph 6.8) and has a small variance with an average confidence interval at 114. The CPU usage has no tendency of significant variation with a mean value of 66 % (Graph 6.9).



Graph 6.9: CPU usage

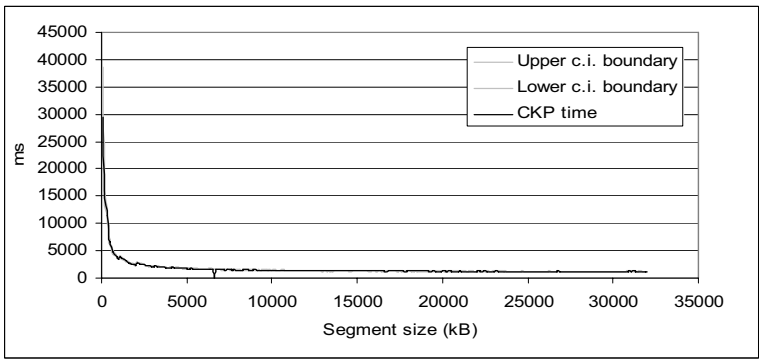
Checkpoint time and number of checkpoints

The number of checkpoints increases from 2 towards 66 checkpoints. This maximum value is reached at a segment size of 27 MB. From this point the graph sinks towards 52 operations, which are reached at a segment size of 31 MB (Graph 6.10).



Graph 6.10: Number of checkpoints

The checkpoint time behaves in a stable, normal way. The variance is very little with an average confidence interval at 87. The form is the same as in the previous test cases but the graph stabilizes with values between 1000 and 1200 ms after 5 MB (Graph 6.11).



Graph 6.11: Measured time to perform a checkpoint

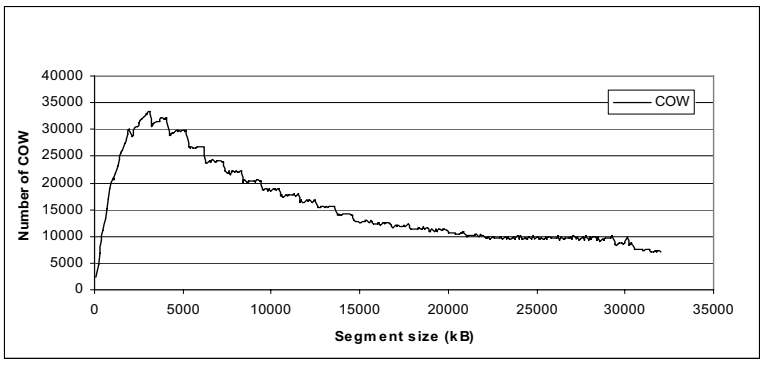
6.3 Uniform load with 25 threads

Graphs in this test case are very similar to runs with fewer threads. The biggest difference is CPU usage which decreases and TPS which is lower. Still, this run shows the same trends in the graphs compared to the earlier runs and has a small confidence interval.

6.3.1 Analysis of graphs

This test case had 25 threads in parallel and all the graphs appear to be normal and represent controlled behaviors. It can be seen as a boundary case, where the system is tested to its limits.

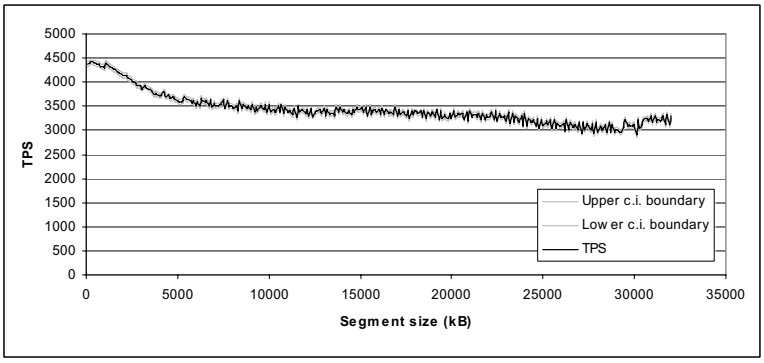
Copy on Write



Graph 6.12: Number of Copy on Write operations

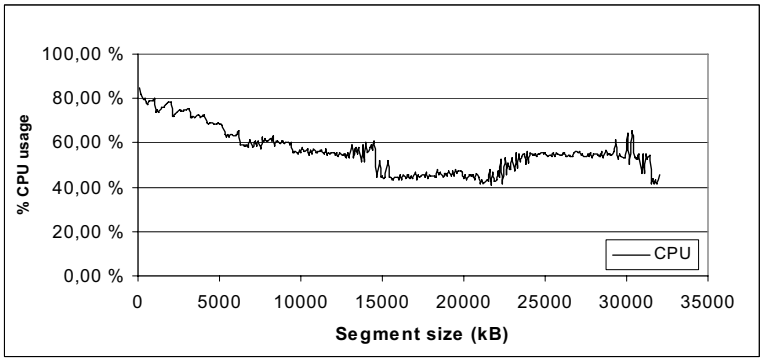
COW time is similar to other runs with different numbers of threads. The load does not show any interesting changes from the other tests. The total number of Copy on Write has a peak on 33444 at 3.1 MB (Graph 6.12). After this peak the graph is asymptotically towards 10000. After 20 MB it flattens out, but has another fall at 30 MB. Here the values decrease towards 7000.

TPS and CPU usage



Graph 6.13: Transactions per Second

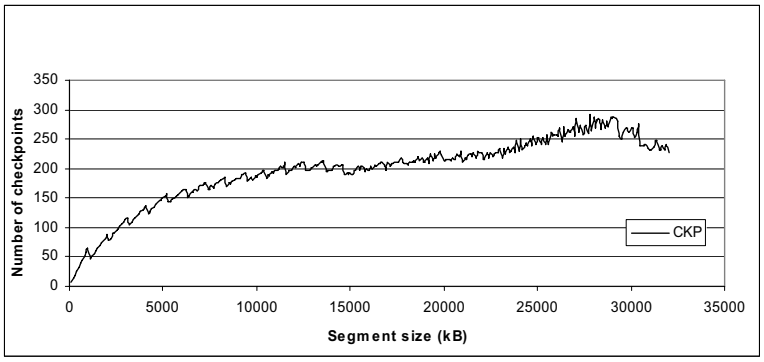
In Graph 6.13 the TPS has a peak at 4412 at 300 kB. From this peak the TPS decreases asymptotically towards 3400. At 22 MB the TPS decreases again towards 3000 before it climbs up again. The TPS has a small confidence interval with an average value at 48 and the variations between neighbor-values are fairly small. CPU usage decreases from 85 % at the beginning, towards 55 % at 15 MB (Graph 6.14). From this point the CPU usage drops to 45 % and stays there to 22 MB, where it increases again.



Graph 6.14: CPU usage

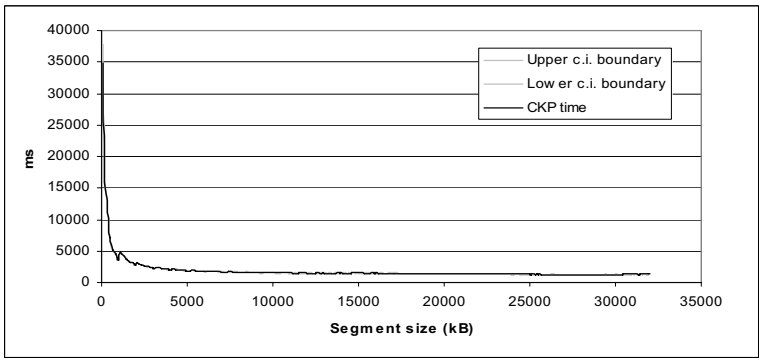
Checkpoint time and number of checkpoints

The number of checkpoints goes asymptotically from 7 to 200. After 15 MB the graph increases towards a peak at 27.8 MB with 292 performed checkpoints (Graph 6.15). After 27.8 MB the graph decreases. The number of checkpoints is very similar to neighbor segment sizes, meaning the graph only has small peaks that overall gives a fairly smooth line.



Graph 6.15: Number of checkpoints

Checkpoint time in Graph 6.16 has a smooth graph that goes asymptotically towards one second and is stable at this value from 5 MB to 32 MB. It also has a very small variance with an average confidence interval at 39.



Graph 6.16: Measured time to perform a checkpoint

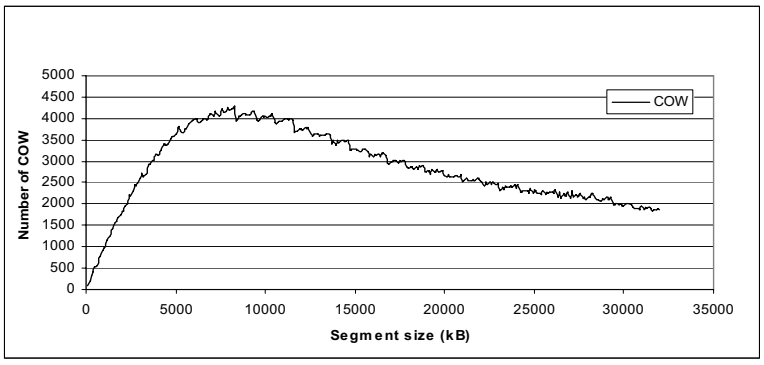
6.4 Distorted load with five threads

Distorted load gives a high access rate for the first segments. In a normal DBMS, some segments are accessed more often than others. This system normally use uniform load, which gives smooth graphs. To see if there is any big difference with distorted load, this test case uses five client threads to access the database distorted.

6.4.1 Analysis of graphs

During this test case with distorted load, the system does not show any radical changes from test cases with uniform load. The biggest change is the Copy on Write peak, which is reached at a different segment size.

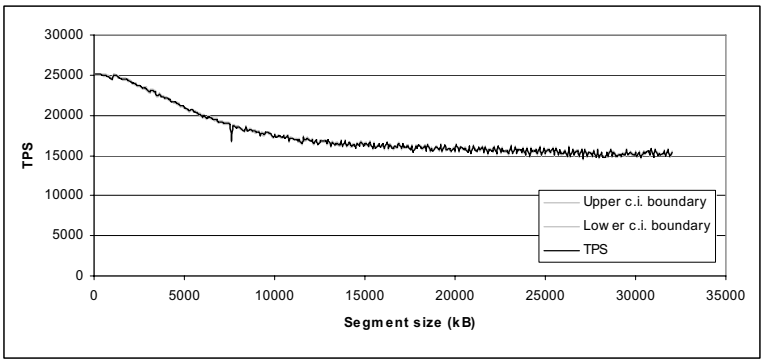
Copy on Write



Graph 6.17: Number of Copy on Write operations

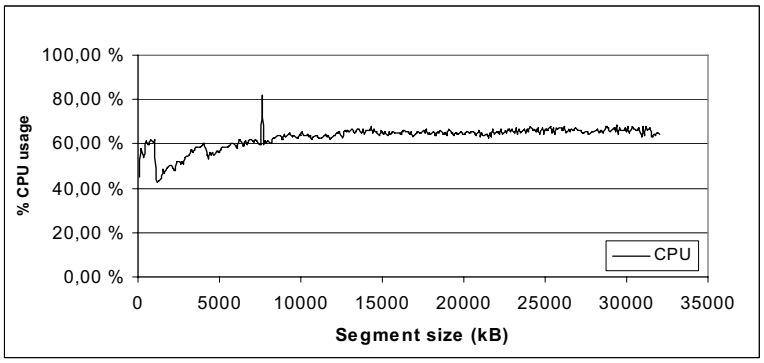
The number of Copy on Write increases from 78 towards a peak at 8300 KB with 4288 Copy on Write operations trigged. From this point it decreases; with a slight asymptotically form, towards 1500. The lowest value after the peak is 1850 at 32 MB. (Graph 6.17)

TPS and CPU usage



Graph 6.18: TPS

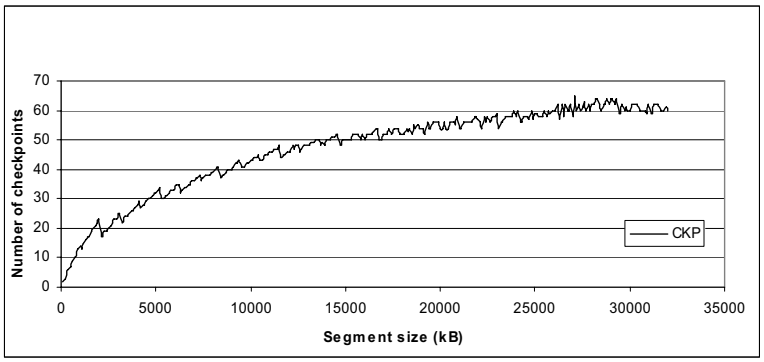
The TPS decreases from 25075 at the beginning, towards 15000 at the end (Graph 6.18). It has a smooth form and only a small drop at 7.6 MB. This small drop at 7.6 MB cannot be seen in any of the other test cases and is thus believed to be a random error. It also has a small confidence interval with an average value at 115. CPU usage is about 60 % in the beginning, but drops to 45 % at 1.1 MB. From this point it increases again towards 60 % with small variance except for the peak at 7.6 MB (Graph 6.19).



Graph 6.19: CPU usage

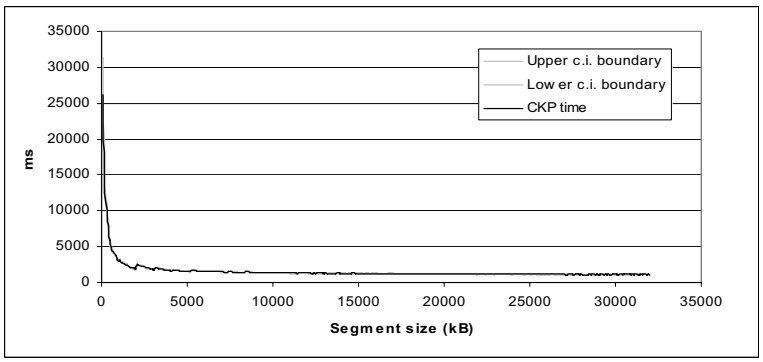
Checkpoint time and number of checkpoints

The number of checkpoints increases from two in the beginning towards 60 in the end. The graph has an asymptotically form and tends to straighten out after 27 MB with 60 checkpoints (Graph 6.20). As in the other cases, the graph is jagged.



Graph 6.20: Number of checkpoints

The checkpoint time shows a stable normal way with a smooth line and a small confidence interval with an average value at 54. It falls from 26245 ms to about 1500 ms at 5 MB. After 5 MB it is very stable (Graph 6.21).



Graph 6.21: Checkpoint time

Chapter 7

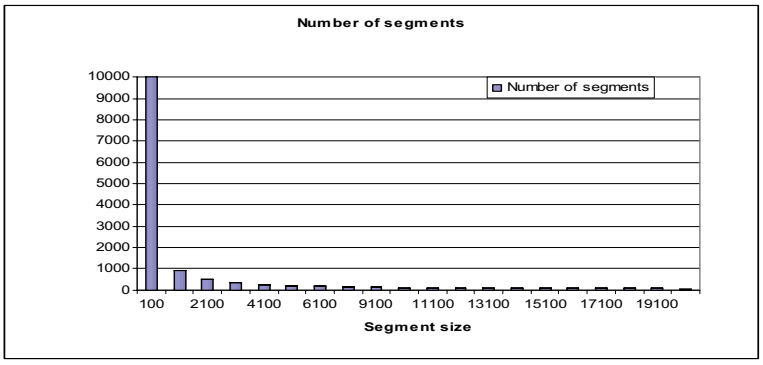
Discussion

In this part of the thesis all the results will be discussed and the focus will be to find relations between the different results. There will be an explanation of trends that appear in the results and why these trends seem to appear. The hypotheses will be discussed and validated and an introduction to different possible weaknesses in the simulator and other possible sources of error will be given. A part of this master thesis has been to create a simulator for simulating use of InfiniBand and RDMA for repair in DBMS. This chapter will give an interpretation of the results and try to conclude trends and optimal settings for such a system. To be able to comment on these results it is important to understand the premises given by the simulation.

7.1 Simulation

InfiniBand and RDMA is a new kind of technology where to-date there has been little research done and the access to necessary hardware is limited. Since this master thesis has been completed over a limited period of time it was decided to create a simulator instead of trying to use InfiniBand and RDMA hardware. When creating a simulator it is very important to be able to simulate a real system load to get the best results. The results from this simulator give mostly predicted results. The simulator has been validated in several ways (Chapter 5.2) to ensure that the code has been written correctly and to perform the operation in an intended way. But even if the simulator is working properly there are a few other factors and design solutions which may have an impact on the results. This section will discuss the effect of these factors and choices and give an explanation as to why these choices have been made.

The first and one of the most important choices was how to simulate RDMA over InfiniBand. When using RDMA over InfiniBand an RNIC is used. This network card contains a processor which is used to remove overhead relating to protocol processing from the CPU. To simulate this the *Java.sleep()* function was used. This function does not use CPU power, so this will give a good indication as to how the transfer happens. When using RDMA one memory area on the destination computer is mapped. This takes some time and will happen for each segment the system is transferring. After a lot of research the setup connection time was decided to be set to 500 μs. The argument for choosing this number was that when using IPoIB the client is not mapping a memory area at the destination before transferring the data. Thus, 500 μs must be the time SDP uses to map the memory area. Since this master thesis is not simulating a three layered web server there is a possibility that this number is inaccurate. The connection time is related to the number of segments. A possible inaccurate estimation of this time will only give the graphs an offset compared to the one presented in this master thesis.



Graph 7.1: Number of segments

As can be seen in Graph 7.1 there is a big drop in the number of segments when the segment size is small. There are 10 000 segments when the segment size is 100 KB. When the segment size is 4 MB the number of segments has been reduced to 250.

When the segment size is 100 KB, 5 seconds is used just in connection time when transferring the whole 1 GB database.

$$10000 * 0{,}5 = 5000ms = 5s$$

When the segment size has increased to 4 MB this time is reduced 40 times to 0.125 s.

$$250 * 0{,}5 = 125ms = 0{,}125s$$

When this model was made it was important to create a model which could simulate a DBMS in the best possible way. But the system had to be simplified many times because of the time estimated for the master thesis. It was decided that the simulation of transaction load was one of the critical aspects of the assignment. Many solutions were tried before the final solution was found (log segments, back-to-back transactions, distorted load and a run with constant 1000 TPS per thread).

When running back-to-back transactions the system had a CPU usage of nearly 100 %. This was found to be a possible source of error, so a function for reducing the CPU usage was introduced. But this function reduces the throughput in the system by putting the client thread to sleep at intervals. The simulator slows down the client threads to get a lower CPU usage and can be seen as an unacceptable interference with the system. The alternative was to use nearly 100 % CPU, which makes the OS managing the CPU in a way impossible for the simulator to control. This was found to be a much larger source of error.

When creating a mockup DBMS it is very difficult to get a correct simulation of all the administrative costs concerning the execution of a transaction. This can have resulted in a simple model to actually get good enough simulation. That is why the values in the results, such as TPS, may not be representative for actual TPS in such a system.

The purpose of this master thesis is not to get exact values but to get graphs which vary and show trends when changing different parameters in the system. For that purpose the simulator is working satisfactorily. When reading the results it is important to remember that when running a simulator with Windows XP there are some parameters that are impossible to control.

The OS used in these simulations uses a time slice of 15 – 16 ms which may have some interference when performing operations. To wait for the different threads to finish *Thread.join()* is used. This is a predefined API which means that the scheduling is handed over to the JVM and is completely out of the simulators' hands.

7.2 CPU usage

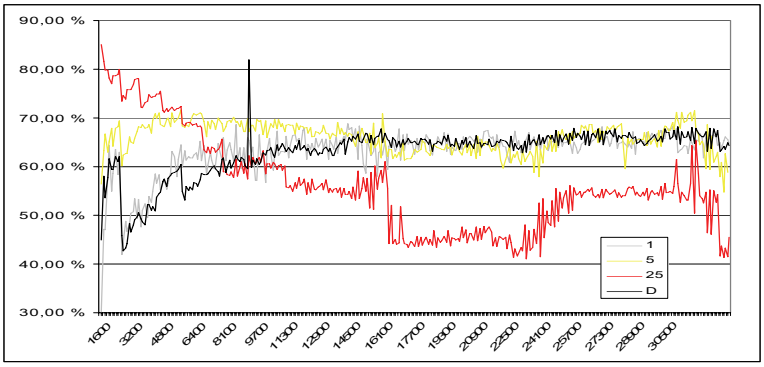
CPU usage in the model has been controlled by running a sleep function periodically, as explained previously. Even if the CPU stays at around 60 %, the graphs show that the usage varies from the beginning, with 100 KB segments, to the end, with 32 MB segments. The analysis from the results show a tendency to vary more, as more client threads are run in the system. For example, in Graph 7.2 the series with one and five threads are smoother than the series with 25 threads. The series with 25 threads has a decreasing tendency from the beginning and a drop between 15 and 22 MB. The CPU results have no direct relation with any of the other results.

There is also a difference before 10 MB compared to after 10 MB. For a few number of threads, no matter the load type, the CPU usage has a local peak. With 25 threads the CPU usage decreases from a high value and down towards 60 %. The instability before 10 MB can be caused by the number of segments that changes dramatically.

The checkpointer has almost no effect on the CPU usage, since it only sleeps the time it takes to perform a checkpoint. It is therefore reasonable to believe that the client threads, using the database, have the biggest effect in this result. These treads use CPU to write data and perform Copy on Write. Since these threads have reduced CPU usage by using a sleep function, it affects the TPS. Normally TPS would be better; the more client threads run in parallel. Since CPU usage is controlled, the system spends more time sleeping, the more threads that run in parallel. TPS will be discussed in Chapter 7.3, and has a strong relation with the CPU usage.

Hypothesis V predicts that the CPU usage will increase if the segment size increases. As can be seen from Graph 7.2, the CPU usage is stable after 10 MB and the hypothesis fails. Number of threads is the factor that affects the CPU usage most.

The hypothesis was put forward before the prototype was constructed. Thus the CPU regulating function was not taken into consideration.

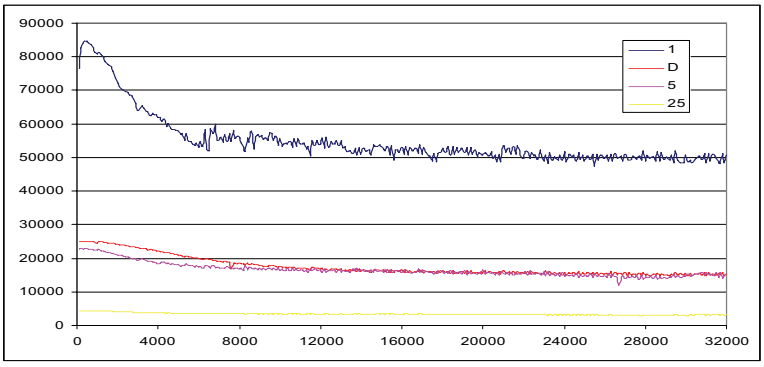


Graph 7.2: CPU usage

7.3 TPS

There are three main factors that affect the TPS; number of segments, Copy on Write operation and CPU usage limit. The effect can best be represented if all the TPS graphs from the test cases are plotted in one new graph. This has been done in Graph 7.3.

To check if the uniform load is a good load simulation the model has been run with distorted load as well. Graph 7.3 shows that the performances in the uniform and distorted test cases are almost equal. The only difference is that distorted load gives a small improvement on segment size under 5 MB.



Graph 7.3: TPS

Graph 7.3 shows that the TPS is low for test cases with many threads compared with test cases with few threads. Normally the TPS would increase with more threads, since it would utilize the CPU better.

In this case the CPU limit will affect the TPS in the opposite way than it normally would, with respect to number of threads. The more threads there are in the system, the more each one will sleep and cause an extra delay on the transaction processing.

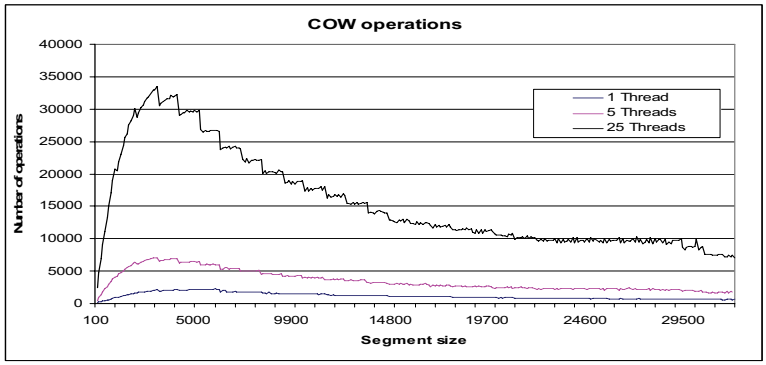
All of the TPS graphs from the test cases show a decreasing performance while using larger segment size. The test case with one thread shows this trend best. It drops from about 80000 in the beginning to about 50000 at the end. This is a difference of 30000 transactions per second and the biggest change is between 100 KB and 10000 KB. The write operation by the client threads are not affected by how big the segments are. In our model, the database size is static and the segment size changes. This causes the number of segments to change. With a small segment size there will be many segments and visa versa. Number of segments has been described in Chapter 7.1 and shows that there are big differences up to 10 MB. This trend can also be seen in the beginning of the TPS graphs.

Another trend that can be seen in the graphs is that every one decreases. This is probably caused by the Copy on Write method. During a Copy on Write, the client threads have to wait until the copying operation is finished. The bigger the segments size is, the longer it takes to copy it and the longer the client threads must wait. The chance to trig the Copy on Write mechanism is also smaller with a large amount of segments. This effect is visual in Graph 7.3. All these graphs decrease, but with less difference the more threads there are. The lowest TPS series, with 25 threads, has almost no changes compared to the series with 1 thread. This is caused by the scaling on the axis and the decreasing effect is better visualized in the analysis (Chapter 6).

Hypothesis I predicts that the TPS will fall, when the segment size increases. From Graph 7.3 it is possible to conclude that the hypothesis is valid. Every test case that has been run in this thesis has a decreasing TPS value with respect to segment size.

7.4 Copy on Write

In this section the Copy on Write operation is evaluated. The results collected from the runs concerning Copy on Write are based on time and number of operations performed. From hypothesis IV it was predicted that the number of Copy on Write operations would increase as the segment size increased. This hypothesis was based on the fact that the probability of hitting a locked segment would increase as the segment size increased. Graph 7.1 shows the significant drop in the number of segments for segments under 10 MB. The results, however, show a different shape than expected (Graph 7.4).



Graph 7.4: Copy on Write operations

The results show that the number of Copy on Write operations reaches a peak at around 3 – 6 MB. From this point the graph falls towards a stable condition at 20 MB. This shows that hypothesis IV fails. To base the conclusion in the hypothesis on the change in probability of hitting a locked segment alone is not enough. What hypothesis IV does not discuss is the time each segment is held by the checkpointer. As the segment size increases, the time taken to transmit the segment also increases. For each segment only one Copy on Write is performed. The next transaction which wants to update this segment is redirected to update the copied segment. This means that at one time the segments will be held for such a long time that the number of Copy on Write operations will fall. But until this point is reached the number of Copy on Write operations will increase, as predicted in hypothesis IV. This means that hypothesis IV is valid for segments smaller that 3 MB.

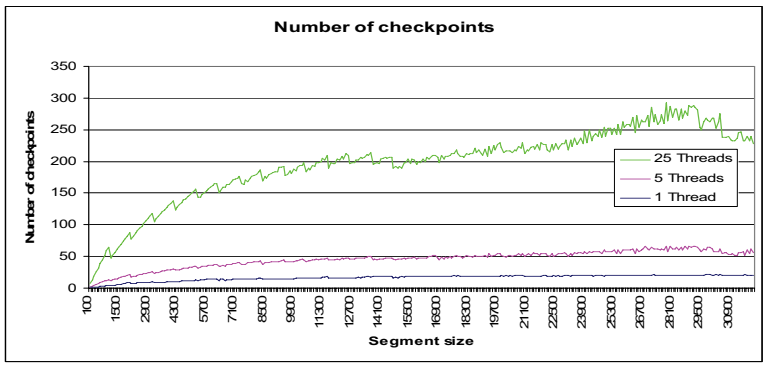
From the peak and down to a stable situation the graph is decreasing in a stair pattern. This pattern is best shown for 25 threads in Graph 7.4. This effect was investigated to find out what could cause this. The research did not give any good answers to that question. The value falls from one segment size to the next. After a lot of research, it has been concluded that this pattern may be due to some OS or JVM-dependent behavior. This possible error was investigated by reading the debug log in detail and running tests with much higher resolution around the breakpoints in the stair pattern. This definite behavior was found and it was not possible to give a 100 % define conclusion as to this behavior.

When looking at the number of Copy on Write operations, it is possible to conclude a trend. From 3 – 6 MB (depending on the number of threads in the system) the number of Copy on Write operations will fall towards a stable value around 20 MB. A test with distorted load was also performed. This test shows the same trend as the other runs (Graph 6.17). In the run with distorted load the peak is relocated to 8 – 9 MB and is due to the limited access area of the database. The probability of hitting a locked segment increases dramatically for parts of the database and decreases dramatically for other parts of the database. The other result from the run concerning Copy on Write is the time each operation takes. This value increases linearly in a stable way (Chapter 6).

Hypothesis III predicts that the transfer rate inside the main memory will stay the same. Graph 5.2, page 51, plots the measured throughput for Copy on Write, and shows that the throughput stays stable around 2 GB/s after 2 MB. This means that hypothesis III is valid.

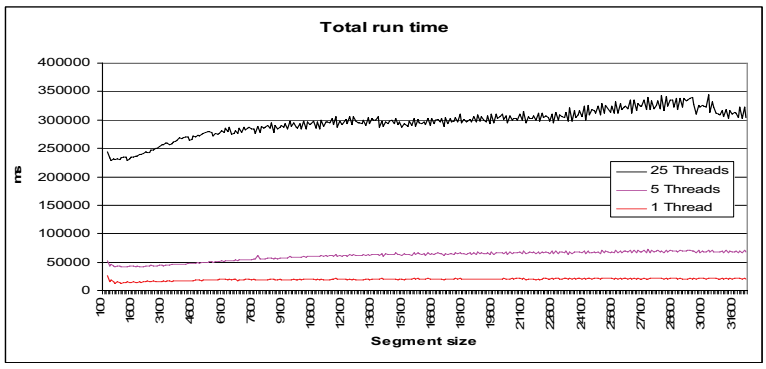
7.5 Checkpointing

There are two different results to analyze concerning checkpointing; number of checkpoints performed and time for performing each checkpoint. First, the number of checkpoints will be discussed. As can be seen in Graph 7.5, all the series show the same shape. The reason they do not look exactly the same is because of the scaling of the graph. Chapter 6 contains more detailed graphs.



Graph 7.5: Number of checkpoints

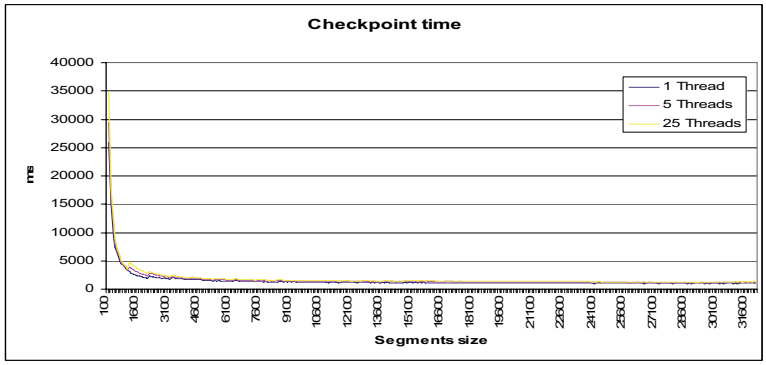
Hypothesis VI predicts that the number of checkpoints will increase if the number of Copy on Write operations increases. As can be seen from Graph 7.5 the number of checkpoints increases despite the fact that the number of Copy on Write operations decreases from 5 MB towards 32 MB. The number of checkpoint operations reaches a peak, but this is at 29 MB and from this point on it decreases towards 32 MB. This peak is believed to be due to some variance expected in empirical testing. There are only two parameters that have an influence on the number of checkpoints, namely the total run-time and the checkpoint time. To show that it is the total number of run-time of the system that has the biggest influence, these values are plotted in Graph 5.6.



Graph 7.6: Total run-time

These two graphs show a close connection between the number of checkpoints and the total run-time for the system. The reason is obvious; the longer the system runs, the more time is available for the checkpointer to take checkpoints.

One of the most interesting results derived from this master thesis is the checkpoint time. As can be seen in Graph 7.7 the checkpoint time is unaffected by the number of client threads introduced to the system. All three runs show a clear trend.



Graph 7.7: Checkpoint time

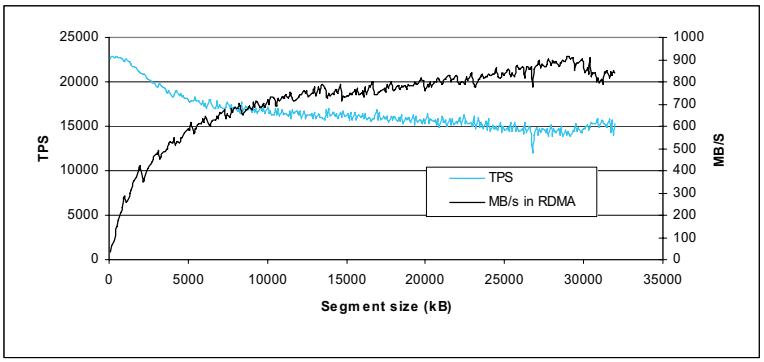
When the segment size gets bigger than 10 MB, there is little effect in increasing the segment size even further. This observation was the whole motivation for the thesis; to find out the max size of segments that would be beneficial to use. When using this graph and comparing it with the TPS graph it is possible to find a solution that will give a short checkpoint time and a high TPS. This will be done more carefully in the conclusion (Chapter 8). It is likely that when the segment size reaches around 10 MB the connection setup time (500 μs) will be such a small part of the total checkpoint time that increasing the segment size even more will have a very little effect. These results show that hypothesis VII is valid, because the checkpoint time decreases when the segment size increases. Compared to Graph 7.1 is it also possible to see the similarities between number of segments and checkpoint time.

7.6 Repair time and throughput

The throughput on checkpoint plays an important role in the repair time. This is calculated by dividing the amount of data to be transferred on checkpoint time. With better throughput, repair time will decrease and availability will be better. One of the factors that affect the repair time most is segment size. The only question is how large this segment size should be. A good TPS is important to give a good service to the client user. With a large segment size the TPS will be low and the system will have a poor performance.

With a clustered DBMS the throughput during a repair of a node reduces the chance of losing a complete fragment. After one node has failed, the system repairs itself in two steps. First, it sets the hot-standby node as the primary node. Then, it creates a new hot-standby node from a spare node and updates the new hot-standby node by copying data from the new primary node. When copying the data during a repair, the system uses the checkpoint technique. Using a large segment size, the throughput on RDMA will increase since the connection time will be insignificant. The total number of segments will decrease and together these two factors will reduce the checkpoint time and increase the throughput.

In Graph 7.8 the throughput and TPS for a test case with five threads has been plotted in the same graph. The form on the graphs is equal in every test case. This was also used to validate hypothesis II which said that there would be better throughput (MB/s) for RDMA if the segment size increased. Graph 7.8 shows that hypothesis II is valid.



Graph 7.8: TPS vs. MB/s

The left axis in this graph is the TPS and the right one is the throughput in MB/s for during one checkpoint. The throughput is dominated by checkpoint time and number of segments, so the lower checkpoint time and the fewer segment, the better throughput. Hence the throughput rate increases with the segment size. The TPS is dominated by trigged Copy on Write operations and segment size. With an increasing segment size, the TPS will decrease and give a lower performance. It is possible to increase the TPS and availability by introducing more nodes in the DBMS cluster. Graph 7.8 shows the result from this model and shows a large difference in the beginning, but both graphs have an asymptotic form that reaches a limit after 10 MB. After this point the series have no large increases or decreases. This point gives a very good throughput, but a low TPS.

7.7 Availability

So how does this throughput affect the availability? Equation 1.4 at page 10 shows the unavailability dependent on the repair intensity ρ_r. If the repair time is small, the unavailability will be small. This can be shown by using Equation 1.4 to calculate the availability depending on the segment size (Graph 7.9).

The repair and recovery intensity in the equation is one divided on repair and recovery time. By replacing the intensity with the time and looking at the case where one node fails, the equation becomes:

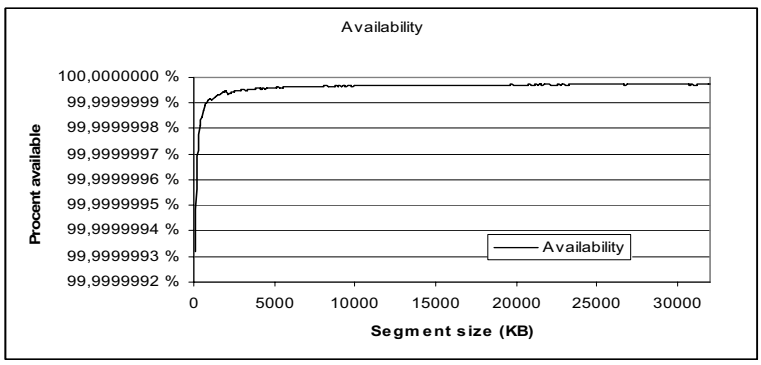
$$U_{NN,on-line} = \lambda_1^2 GN\Gamma_{rr}\Gamma_{ra} \qquad (7.1)$$

The checkpoint time represents the repair time and the other parameters are set to reasonable values. λ_n for one node is set to four failures a year and the chance for getting a double fault (G) is two. The time to repair a node after double fault is set to one hour and the cluster consists of two nodes. Availability is calculated by subtracting one from unavailability and now the equation looks like this:

$$A = 1 - U_{NN,on-line} = 1 - \left(\left(\frac{4}{8760} \right)^2 * 2 * 2 * 1 * CkpTime \right) \qquad (7.2)$$

CkpTime = Checkpoint time in hours.

In Graph 7.9 the availability is plotted and it shows a good resemblance to the checkpoint time. It rapidly increases and has no significant changes after 10 MB.



Graph 7.9: Availability for a two node system

At 10 MB the availability is 99.9999999677943 % and with Equation 1.2 at page 6, it shows that this is class 9 availability. Tang, Kumar, Duvur and Torbjørnsens article on availability on Sun's Java System Application Server system shows the availability in HADB today.

This system uses HADB as Database Redundancy Units and has a repair time on 12 minutes in 2004 [TKD04]. 12 minutes is used to transfer 1 GB from one node to another which is the same case simulated in this master thesis. With Equation 7.2 and 1.2 the availability of this system is 99.9999833197807% which is class 6. Class 6 is a very high availability but three classes below class 9. This shows the significant improvement that can be achieved with use of InfiniBand and RDMA. With this technology the data transfer during repair will no longer be the bottleneck of DBMS repair. In practice class 9 may not be achievable since there may be other factors affecting the repair time negatively, but it shows the important role of InfiniBand and RDMA in DBMS clusters in the future.

7.8 Source of errors

When these results have been discussed and analyzed it is important to try and define possible sources of error. This chapter tries to define and explain the significance of such sources of error.

When presenting such data as produced in this master thesis, it is important to scale the graphs in a suitable way, and not just scale the graphs in such a way that they give something more like the predicted results. For this thesis the detailed graphs can be found in the result chapter (Chapter 6). The discussion chapter has the runs with different amount of threads being plotted together in one graph. This is to give an impression of the difference or likeness between all the runs. Such a plot also gives a good verification on the stability of the system when the graphs are showing the same trends.

During simulation in this assignment there have been a lot of other active processes beside the DBMS. Before executing each test case, all the unnecessary processes have been killed. The remaining processes may have affected the simulation in some way. Real-world runs will also have other processes in parallel affecting the results. Therefore this source of error will be considered insignificant.

All the tests have been performed several times to check for random errors and the conclusion has been that the simulation has given the same results each time.

7.9 Summary of hypotheses

During the last chapter there has been a discussion over the result produced by this master thesis. Chapter 4 presents the hypotheses validated by these results. This chapter aims to give a short summary of the results concerning the hypotheses. Because the arguments which validate the different hypotheses are spread around in the report a table is created to give a better overview of the results. Table 7.1 shows the results for the different hypotheses and states which chapter contains the argument for the result of the hypotheses.

Hypotheses	Result	Chapter containing argument
I.	Valid	7.3
II.	Valid	7.6
III.	Valid	7.4
IV.	Partially valid	7.4
V.	Invalid	7.2
VI.	Partially valid	7.5
VII.	Valid	7.5

Table 7.1: Hypotheses results

Chapter 8

Conclusion

This thesis has given a description of the behavior in a highly available DBMS using RDMA and InfiniBand. A mockup DBMS and a simulation of RDMA and InfiniBand has been used to collect data to perform analysis on. The main focus has been on TPS, CPU load, checkpoints, Copy on Write operations and repair time.

The work done in this thesis explored a novelty value for the use of InfiniBand and RDMA in DBMS clusters. The results showed potentials for a huge improvement concerning availability for such systems. When simulating a system with such state-of-the-art technology there are several possible pitfalls, this thesis have tried to discover these and given possible solutions. If this thesis was taken further with a complete implementation together with HADB, it would increase the availability. This may change the way of thinking concerning DBMS cluster design by reducing the number of error handling operations. This show the utility value discovered in this thesis.

The methodic reliability in this thesis can be discussed in many ways. Since a simplified DBMS is used and the actual behavior of RDMA is unclear, there are some uncertainties concerning the results. These uncertainties have been discussed to give a better understanding of the premises for this thesis. To assure that the simulator worked according to plan the source code was tested and validated in several ways. With the premises given by professor Bratsberg and Sun Microsystems AS, this thesis gives an answer to the question: *Will large segments in DBMS clusters give an improved performance when using InfiniBand and RDMA?* This was the origin for making this thesis.

For TPS, checkpoints and repair time there are no big changes in the results after 10 MB. This can be related to number of segments, which has the same form. 10 MB, or 1 % of the database size, has 100 segments. The difference in number of segments from the smallest segments size point (100 KB) is 9900 and from the largest point (32 MB) is 21. After 10 MB, the simulation shows no radical changes in the results.

The Copy on Write operations has a different graph form than all the other results. It has a peak which is dependent on the load. If there are some segments that are used very often, for instance log segments, the load will be distorted. In this implementation distorted load shows a peak at 8 – 9 MB and with uniform load the peak will be at 3 – 6 MB.

Since the simulator is implemented with a CPU load limit, the results are affected. If the system did not have this function the load would always be 100 % and not interesting at all. The control of which sequence things are performed in would also be almost none. The cost of implementing this function has been lower performance with an increased number of threads. They still show the trends and forms that would be normal, but the values are lower. The CPU load limit was successful for its cause.

Doing a mockup DBMS and simulation of RDMA and InfiniBand has given satisfying results. It shows trends and behaviors that were not foreseen before implementation. The value range of the data may not be representative for a real DBMS, but the graph form shows the trends. This was the goal for this thesis and this result can give an indication of what type of segment size is beneficial to achieve a high availability.

Availability in Sun's HADB is the background for this thesis. With state-of-the-art technology and optimal configuration the availability will be improved. This thesis has given an indication that the configuration of segment size should not be more than 1 % of the database size. With use of InfiniBand and RDMA using this configuration and physical repair, the availability increased from class 6 to class 9.

Due to time limitations and lack of available hardware and software, this thesis has not been tried out on a real DBMS, RDMA and InfiniBand technology. There are some changes that may give a more realistic result, as described in the next chapter.

Chapter 9

Further work

Further work will be to implement the system with physical use of InfiniBand and RDMA. A solution that implements InfiniBand physically will give a better approach to the problem concerning connection setup time. When implementing InfiniBand it is necessary to evaluate all the needed hardware. This can both be expensive and time consuming. As there has not been much previous research on this subject, it would be very difficult to choose feasible hardware. A part of the further work would be to collect and test different kinds of InfiniBand hardware to determine which would be most feasible for the use of checkpointing in a DBMS.

After the hardware is chosen the most efficient way to implement RDMA must be found. Tests need to be conducted to find the optimal implementation for RDMA over InfiniBand. As can be read in Chapter 3.1 others have failed when trying to implement RDMA over InfiniBand because of the degree of difficulty.

When these two subjects have been investigated there would be a steady basis to start the integration of both hardware and implementation of RDMA. There are a lot of challenges concerning this integration. It will demand a lot of time to research to find optimal solutions since it is a state-of-the-art technology. The projects will need sturdy financial frames because of the hardware. When implementing such a system there is also a big risk that the project never will be completed within the planed time scope because of all the unforeseen factors that will appear.

After the integration, it is possible to improve the research even further by implementing HADB as the DBMS. This will give a realistic transaction load and CPU usage. Both NTNU and Sun Microsystems have a lot of resources on HADB which should make it possible to implement InfiniBand as a transfer mechanism for checkpoints during repair without any large complications.

Implementing the complete system makes it possible to get real results. These results would not have been affected by CPU limitations or inaccurate connection setup time for RDMA. It will also be possible to measure the improved performance in the system by measuring the performance according to some benchmark standard. Hopefully, this complete implementation will show the same trends as the simulation performed in this master thesis.

Chapter 10

Glossary

The words and phrases presented in this glossary that are not explained in the thesis is collected from http://en.wikipedia.org

A window in TCP Sliding Windows protocol	In transmit flow control, sliding window is a variable-duration window that allows a sender to transmit a specified number of data units before an acknowledgement is received or before a specified event occurs.
Back end server	A back-end server is a server with a standard configuration. The term "back-end server" refers to all servers in an organization that are not front-end servers after a front-end server is introduced into the organization.
Back-to-back transactions	Transactions are preformed in serially and not parallel order. With no wait time.
Bcopy	An internal copy is made before sending data over the network.
Benchmarks	A benchmark is a point of reference for a measurement. The term presumably originates from the practice of making dimensional height measurements of an object on a workbench using a graduated scale or similar tool, and using the surface of the workbench as the origin for the measurements.

Clustra	Clustra is a telecom database prototype developed to run on standard workstations interconnected by a switch.
Context switches	A context switch is the computing process of storing and restoring the state of a CPU (the context) such that multiple processes can share a single CPU resource. The context switch is an essential feature of a multitasking operating system. Context switches are usually computationally intensive and much of the design of operating systems is to optimize the use of context switches.
COW	Copy on Write. The principles that you can efficiently share as many read-only copies of an object as you want until you need to modify it. Then you need to have your own copy.
CPU	Stands for Central Processing Unit, a programmable logic device that performs all the instruction, logic, and mathematical processing in a computer.
DBMS	A database management system (DBMS) is a computer program (or more typically, a suite of them) designed to manage a database, a large set of structured data, and run operations on the data requested by numerous users
DDR	Double Data Rate (memory)
DMA	Direct memory access (DMA) allows certain hardware subsystems within a computer to access system memory for reading and/or writing independently of the CPU.
HADB	High Availability Data Base, A product from Sun Microsystems

HCA	Host Channel Adapter
I/O operations	Services that provide access to shared input/output devices and to the global data structures that describe their status. I/O operations open and close files and devices, read data from and write data to devices, set the state of devices, and read and write system data structures.
InfiniBand	InfiniBand is a high-speed serial computer bus, intended for both internal and external connections
Latency	Latency is the transit time through a digital process, from input to output. It is a minimal, and usually undesirable, delay.
MTTF	The mean time expected to the first failure of a piece of equipment. It is a statistical value and is meant to be the mean over a long period of time and large number of units.
MTTR	Mean Time To Repair. The average time required to repair a failure. Automated fault isolation techniques, including automatic fault bypassing, have reduced this measurement of system recovery time.
NFS	Network File System (NFS) is a protocol originally developed by Sun Microsystems in 1984 and defined in RFCs 1094, 1813, (3010) and 3530, as a file system which allows a computer to access files over a network as easily as if they were on its local disks.
NIC	Network Interface Car

OSI model	The Open Systems Interconnection Reference Model (OSI Model or OSI Reference Model for short) is a layered abstract description for communications and computer network protocol design, developed as part of the Open Systems Interconnect initiative. It is also called the OSI seven layer model.
PCI	The Peripheral Component Interconnect standard (in practice almost always shortened to PCI) specifies a computer bus for attaching peripheral devices to a computer motherboard.
PCI-X	Peripheral Component Interconnect Express. New PCI standard with much higher transfer speed than PCI.
QDR	Quad Data Rate
RAID	In computing, a redundant array of independent disks (more commonly known as a RAID) is a system of using multiple hard drives for sharing or replicating data among the drives.
RDMA	Remote Direct Memory Access (RDMA) is a concept whereby two or more computers communicate via Direct Memory Access directly from the main memory of one system to the main memory of another.
Response time	In telecommunication, response time is the time a system or functional unit takes to react to a given input.
RNIC	RDMA enabled NIC

RPC	A Remote Procedure Call is a protocol that allows a computer program running on one host to cause code to be executed on another host without the programmer needing to explicitly code for this.
SDP	Sockets Direct Protocol
State-of-the-art technology.	The newest technology.
TCA	Target Channel Adapters
TPS	Transactions Per Second
UDP	User Datagram Protocol
ULP	Upper Layer Protocol
Zcopy	No (Zero) copy is made internally before transferring data over the network.

Chapter 11

Bibliography

All preliminary research used as background for this thesis is referred to in this chapter. Reference to web pages is dated with the year presented in the web page. If the web page does not present any date or year, the time when the page was visited is presented.

[Ali01] Anasstasia Ailamaki, *Recover with ARIES,* http://www-2.cs.cmu.edu/afs/cs/academic/class/15721-f01/www/lectures/recovery_with_aries.pdf , 2001

[Bal03+] Pavan Balaji, et al. *Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial?,* http://nowlab.cis.ohio-state.edu/publications/conf-papers/2004/balaji-ispass04.pdf, 2003

[Bal04+] Pavan Balaji, et al. *Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial?* http://nowlab.cis.ohio-state.edu/publications/tech-reports/2004/balaji-ispass04-tr.pdf, October 2004

[Cal03+] Brent Callaghan, et al. *NFS over RDMA,* http://delivery.acm.org/10.1145/950000/944753/p196-callaghan.pdf , August 2003

[Cha02] Wai Tik Chan. *Take the new I/O in JDK 1.4 for a test drive,* http://builder.com.com/5100-6370-1049567.html, December 2002

[Etv04] Etvcookbook. *Latency*, http://etvcookbook.org/glossary/, May 2004

[GR93] Jim Gray, Andreas Reuter. *Transaction Processing*, IMorgan Kaufman Publishers, 1993

[Gre02] Paul Grenyer. *A short article written for the ACCU on the principle of Copy On Write,* http://www.paulgrenyer.co.uk/articles/copy_on_write.htm, January 2002

[GS91] Jim Gray, Daniel P. Siewiorek. *High-availability computer systems.* http://ieeexplore.ieee.org/iel1/2/2779/00084898.pdf?tp=&arnumber=84898&isnumber=2779, September 1991

[Hal02] Tom. R. Halfhill. *What you should know about InfiniBand*, http://storagemagazine.techtarget.com/magItem/1,291266,sid35_gci821124_idx2,00.html (Required registration!), May 2002

[HK04] Steffen Holthe, Jan Steinar Kvilesjø. *Transactions in databasesystems, http://www.kvilesjo.net/a2004ntnu.pdf*, December 2004

[ITA04] InfiniBand Trade Association, *InfiniBand SM Trade Association Announces Completion of 1.2 Specification,* http://www.infinibandta.org/newsroom/Spec_announcment_Sep_8_04.pdf, September 2004

[JM03] Lars E. Jonsson. William R. Margo. *Comparative Performance of InfiniBand Architecture and Gigabit Ethernet Interconnects on Intel Itanium 2 Microarchitecture-based Clusters,* http://www.dynamore.de/download/eu03/papers/K-II/LS-DYNA_ULM_K-II-13.pdf, 2003

[Kim04a] Ted H. Kim. *Brief History of InfiniBand: Hype to Pragmatism,* http://blogs.sun.com/roller/comments/RandomDude/Weblog/history_hype_to_pragmatism, July 2004

[Kim04b] Ted H. Kim. *So what is InfiniBand?* http://blogs.sun.com/roller/comments/RandomDude/Weblog/so_what_is_infiniband, June 2004

[Lon03] Byran Longmire. *The InfiniBand Architecture for 2003 and 2004,* http://www.techonline.com/community/ed_resource/feature_article/24364, 2003

[Man01] Kevin T. Manley *High Resolution Timer,* http://www.fawcette.com/archives/premier/mgznarch/javapro/2001/08aug01/km0108/km0108-2.asp, August 2001

[Pan04] D. K. Panda. *Design of Scalable Data – Center with InfiniBand, http://nowlab.cis.ohio-state.edu/projects/data-centers/*, October 2004

[Pen02] Odysseas Pentakalos. *An Introduction to the InfiniBand Architecture*, http://www.oreillynet.com/pub/a/network/2002/02/04/windows.html, April 2002

[Pet02] Zachary Nathaniel Joseph Peterson. *Data placement for Copy on Write using virtual contiguity,* http://www.znjp.com/papers/peterson-UCSC-MS02.pdf, September 2002

[Pic03] Picture. http://www.trireme.com/whitepapers/process/adoption/rup.gif, 2003

[Pin02a] Jim Pinkerton. *The Case for RDMA,* http://www.rdmaconsortium.org/home/The_Case_for_RDMA020531.pdf, May 2002

[Pin02b] Jim Pinkerton. *Sockets Direct Protocol v1.0,* http://www.infinibandta.org/events/past/spring2002/2_Sockets_Direct_Protocol.pdf, 2002

[Pin03] Jim Pinkerton. *Socket Direct Protocol v1.0 RDMA Consortium,* http://www.rdmaconsortium.org/home/SDP_tutorial_v1.0d.pdf, October 2003

[Ras91] Shelly Rasmussen. *An Introduction to statistics with data analysis*, Thompson information/publishing group, 1991

[Ric01] Robert Richmond. *InfiniBand: Next Generation I/O*, http://sysopt.earthweb.com/articles/infiniband/index.html , January 2001

[Rou02] Vladimir Roubtsov. *Profiling CPU usage from within a Java application,* http://www.javaworld.com/javaworld/javaqa/2002-11/01-qa-1108-cpu.html, November 2002

[Sla05] Slashdot. *Linux Kernel 2.6.11 Released.* http://linux.slashdot.org/article.pl?sid=05/03/02/1331245&from=rss, March 2005

[SMI04] Sun Microsystems Inc. *Java NIO API,* http://java.sun.com/j2se/1.5.0/docs/api/java/nio/ByteBuffer.html, 2004

[SMS02] Heidi Scott, Patric Martin, Bernie Shiefer. *A Study of the Impact of Direct Access I/O on Relational Database Management Systems,* http://portal.acm.org/ft_gateway.cfm?id=782125&type=pdf , 2002

[Str03] Are Joachim Strande. *Testspesifikasjon*, http://www.kongstud.hibu.no/d09-2003/ (Username and password required), May 2003

[TKD04] Dong Tang, Dileep Kumar, Sreeram Duvur, Øystein Torbjørnsen. *Availability Measurement and Modeling for An Application Server.* http://csdl.computer.org/comp/proceedings/dsn/2004/2052/00/20520669abs.htm, August 2004

[Tor95] Øystein Torbjørnsen. *Multi-Site Declustering Strategies for Very High Database Service Availability.* Trondheim: University of Trondheim, 1995

[Tra03] Greg Travis. *Getting started with NIO*, http://www-106.ibm.com/developerworks/edu/j-dw-java-nio-i.html, July 2003

[Web01] John Webster. *Fibre Channel vs. IP vs. InfiniBand,* http://is.pennnet.com/Articles/Article_Display.cfm?Section=Articles&Subsection=Display&ARTICLE_ID=107327 (Required registration!), July 2001

[Woo04] Gordon Wood. *Guides to the Medical Literature.* http://www.fammed.ouhsc.edu/RobHamm/UsersGuide/define.htm, 2004

Chapter 12

Appendix

Appendix I **Work flow**

Appendix II **Source code**

Appendix III **Sample of result data**

Appendix IV **Sample of debug log**

Appendix I

Work flow

Chapter 1

Work flow

The main goal for the software system is to be able to simulate a real database system in a satisfying way. The simulator will be used to find out more about the benefits and disadvantages of using InfiniBand in database clusters. Based on the theoretical information from preliminary research there is likely to believe that InfiniBand will give a huge advantage when transferring large amounts of data. The disadvantage will be that when transferring a large amount of data, the same amount of data will have to be copied internally inside the memory because of Copy on Write. The challenge will be to find which segment size that will give the best overall performance. The optimal solution will give a high transfer rate over InfiniBand and high TPS with a stable CPU usage. To be able to get these results a short checkpoint time and an efficient Copy on Write algorithm is needed.

1.1 The software development process

The development process was planed in detail from the beginning of the project. In this plan a schedule for every activity in the project period was defined. This resulted in a large plan with a lot of small activities (Figure 1.1).

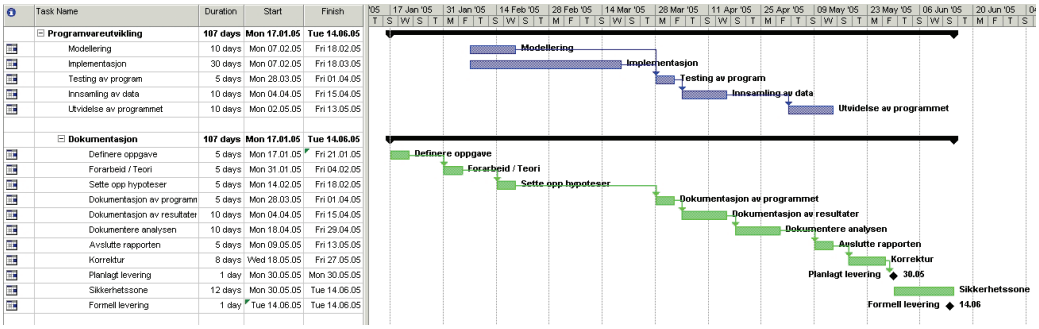


Figure 1.1: Project plan

Before each meeting an evaluation of the plan was made to see if there had to be made any adjustments. This work cycle resulted in a continuously updated project plan at all times which made it able to make any corrections in the work needed to finish the project safely in the end of May.

The activities can be divided into three parts, design, implementation and test. In parallel with these activities a continually update of the project report has taken place. These tree parts were performed in an iterative way. Iterative work means that the different phases are processed with a different accuracy. Figure 1.2 shows the amount of work estimated in the different part of the project. Each of the dotted lines represents a new phase. In new terminology this kind of development is also called agile software development.

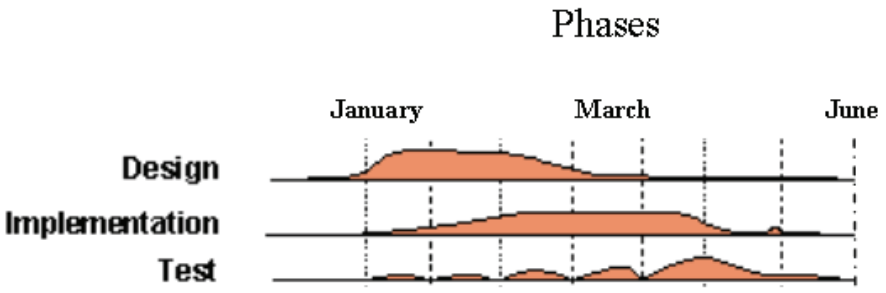


Figure 1.2: Development process [Pic03]

When developing software using agile development methodology it is possible to make changes to the design even late in the project. This makes the whole project much less vulnerable to changes and the chance of success increase considerably.

1.2 Design

In this phase demands concerning the software had to be detected. This was done during meetings with the employer, Sun Microsystems and together with the teaching supervisor at NTNU. In the beginning there were some problems defining the exact goal of the project. This was easily solved when a clear definition of the purpose for the project came from Sun Microsystems. To give the collaborating partners the best insight in the working process a graphical representation of the design phase was used. Rational Rose was used to make these graphical representations. This was done by developing class diagrams, sequence diagrams and activity diagrams to mention a few.

This work also gave a better chance to detect possible problems and difficulties in an early stage. As a part of the preparation to the project some issues considered to be possible pitfalls where detected. To remove these pitfalls from the project they where implemented as prototypes to prove that the implementation was possible. This resulted in a solution to all of the possible pitfalls detected in the start of the project. Some of the possible pitfalls were:

- High resolution timer
- Measurement for CPU usage
- Managing multiple threads (starting, stopping, naming)
- Make the program wait until all the threads have finished.

For a description on how they where solved see Chapter 5.3.1. Figure 1.3 shows a class diagram and shows the relations between all the different classes in the system. Each class is represented by a box containing attributes and operations. This diagram is very useful when trying to explain the system to other people. It gives a complete overview over the system with functionality and was used to make all the parts involved in the project agree on the solution.

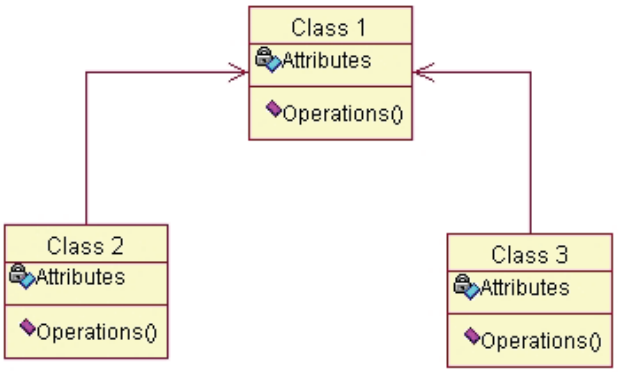


Figure 1.3: Class diagram

When all parts agree on the class diagram the work continues with developing the functionality even further. This was done by using sequence diagram (Figure 1.4). This kind of diagram shows how a single object works crosswise the different classes. This is a good way to detect which operations are needed in each class. Sequence diagrams make the explanation of thread functionality a lot easier.

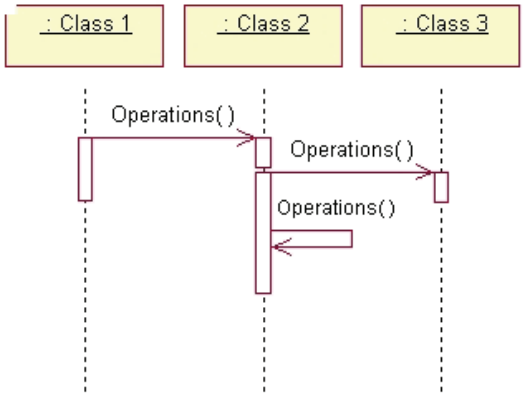


Figure 1.4: Sequence diagram

When the sequence diagram is approved by all the partners and the developers, an activity diagram can then come in handy (Figure 1.5). This diagram explains the functionality to an algorithm in detail. This diagram helps the developers to detect failures in the algorithms before they implement them. This diagram also functions as a recipe when the programmer is implementing the algorithm.

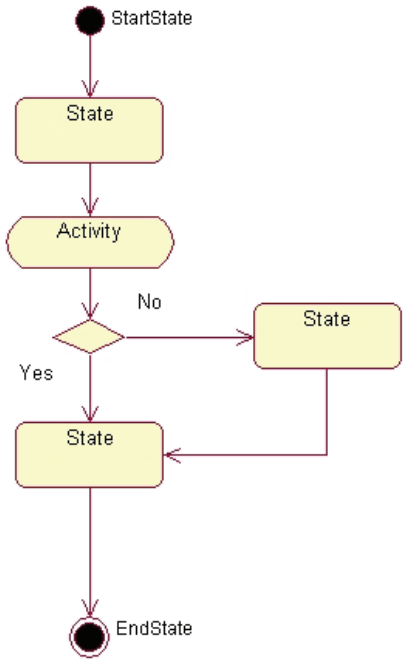


Figure 1.5: Activity diagram

1.3 Implementation

The essence of the implementation process is to transform the design diagrams into actual java code (Figure 1.6). Based on the class diagram Rational Rose can create classes with operations and attributes. This makes it very easy to start the implementation process.

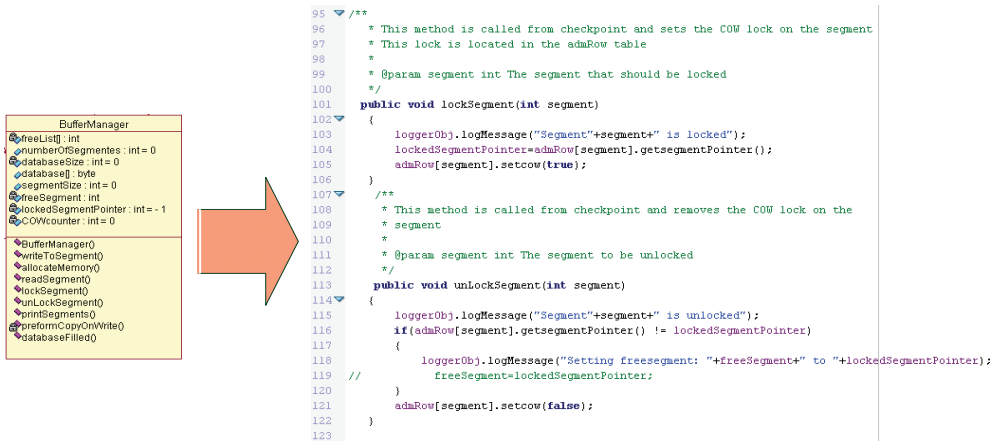


Figure 1.6: Essence of the implementation phase

The first part of the implementation process was used to implement all the classes, operations and attributes from the design phase. Then all the different algorithms where implemented. Since almost every difficult algorithm was documented in detail in the design phase this was pretty a straight forward job. The rest of the implementation job was to implement log function and to check that every part of the system worked as planed. There was made many tests to see how the RDMA and the transactions load worked. In Figure 1.7 a part of the debug log is shown. This log was used to log every operation in the system. In this way we where able to control whether the system performed the right operations at the right time or not.

```
18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(true)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: CopyOnWrite is activated for the3th time

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping
```

Figure 1.7: Debug log

When running the system with the debug log turned on it is important to remember that this will give a significant impact on the results since all the information has to be written to file. That is why none of the final results were collected from runs with the debug log activated. For each run it was taken a lot of different measures, these measures where printed into separate result files and added to a common shared result file. This file contained a number of measurements made during the run. The common shared result file contained the same information but in a different way. Results were added to this file. This function made it possible to make 100 runs with different segment sizes, and then all these 100 results were added into this file. These results can then be imported to MS Excel for a graphical representation of the results (Figure 1.8).

24500	5020.6998008176515	194	7689	1034.0293129147642	0.9827991974885351	11.287971166694655
25000	5072.144877385912	185	7295	1070.729844068815	0.9870595840540046	11.727230479490848
25500	4891.495217926907	192	7452	1069.4862834410499	0.9929159775179643	12.516568952275422
26000	5081.785113300787	190	7181	1042.1938995502376	0.9901944588907402	11.963333710166323
26500	5064.783691021553	196	7209	1014.6629701707698	0.9894372775319222	12.000623713028055
27000	5151.384605848632	187	6851	1043.1323168791694	0.9939820260435533	12.090233374787424
27500	5024.1946569295615	194	6963	1030.2339231364328	0.9931869595419297	12.659761672521663
28000	4932.3145730234255	194	6757	1048.9853599814942	0.9929761023632696	13.665655717851964
28500	4860.279297198973	196	6840	1053.7236492085988	0.9893287695560347	13.785325940531234
29000	5063.860904014198	193	6527	1027.284205187758	0.992970076429628 5	13.322617093768502
29500	5090.203062992185	200	6588	986.8638650614299	0.9879331808296266	12.91023111179607
30000	5122.320986308999	191	6249	1030.1400481834967	0.9907233876988069	13.530178964959598
30500	5003.542164696847	199	6244	1014.4216100819552	0.9895906807660841	14.431056548842824
31000	4828.472838546543	196	6264	1060.4983500836888	0.9924677819667934	15.32433962955413
31500	4900.766635354763	202	6251	1015.2831274446967	0.9919778010859792	15.007005913245123
32000	5131.455193798015	193	5894	1018.5835558018507	0.9909784593029661	14.397858735194603

Figure 1.8: Extract of the common shared result file

These graphs were an important part of the analysis and conclusion made in this thesis.

1.4 Test

The test phase has been performed continuously throughout the whole project. All parts have been individually tested before activated in the whole system. When the system was up and running the debug log was used to detect possible errors. During simulation all results are stored and used to go back and find possible beneficial changes. Since no one has done this before is a large part of the project to find parameters which will give believable results, which is why the system is continuously tested. To make sure the project is on the right track, it is important to start the testing early in the project. To assure that the system have the highest possible quality and standards it has to be validated and verified for every step in the development process. Validation means to make sure that the right kind of system is build, in order to what the customer wants [Str03]. Verification means to make sure that the system work according to the plans.

1.4.1 Testing

Testing can be divided into two different phases, static testing and dynamic testing. Static testing is inspection and verification of documents and code. This will detect typing errors and deviation from guidelines given for the project.

Dynamic testing is testing of the whole system or parts of it. There are two types of dynamic testing, white box and black box testing. White box testing (Figure 1.9) require insight to the internal structure that is about to be tested. During this type of dynamic testing the values are followed thought the whole system. In this project, debug logging is used for this purpose. Black box testing (Figure 1.10) is when only input and output are shown. What is happening inside the system is irrelevant to the tester. Every time something is sent in to the system the results are compared with expected results.

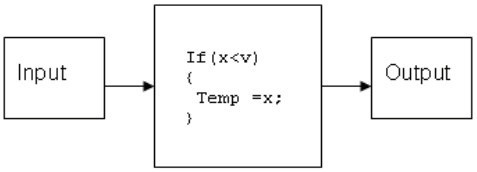


Figure 1.9: White box

Figure 1.10: Black box

It is important to perform regression testing for each new iteration. This means going back and tests something from earlier iterations (Figure 1.11). This is to assure that things still work with the new code integrated, and that there are no new errors introduced. To test this, some selected tests are chosen to be performed once more.

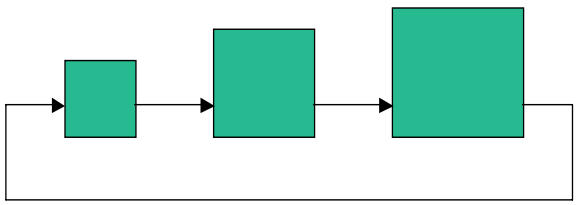


Figure 1.11: Regression testing

Appendix II

Source code

Class DataBaseControl

java.lang.Object
  └ **DataBaseControl**

All Implemented Interfaces:
CPUUsageThread.IUsageEventListener

public class **DataBaseControl**
extends java.lang.Object
implements CPUUsageThread.IUsageEventListener

Constructor Summary

DataBaseControl()	This is the constructor for this class.

Method Summary

void	**accept**(SystemInformation.CPUUsageSnapshot event) Function required by CPUUsageThread.IUsageEventListener
void	**addResults**(java.lang.String s, double res) This method makes it posible for the diferent threads to add their result to one commom string
void	**addResultToExcel**(int number, double result) This method collects results from concerning TPs, checkpoints and COW operations
static java.lang.String	**calculateConfidenceInterval**() This method calculates the confidence interval
static void	**main**(java.lang.String[] args) This is the main method.

```
import com.vladium.utils.*;
import java.io.FileInputStream;
import java.util.Properties;
import java.io.*;
import java.util.*;
import java.text.*;
/**
 *
 * <p>Title: DBMS</p>
 *
 * <p>Description: Creates DB, checkpoint and transaction objects.
 * To run the program you need to go to the class directory and execute this command:
 * java -XX:MaxDirectMemorySize=1G DataBaseControl
 * This is to overwrite the maximum amount of memory to allocate
 * </p>
 *
 * <p>Copyright: Copyright (c) 2005</p>
 *
 * <p>Company: NTNU</p>
 *
 * @author not attributable
 * @version 1.0
 */
public class DataBaseControl implements CPUUsageThread.IUsageEventListener
{
   private static Checkpoint checkpointObj;
   private static TransactionController transactionControllerObj;
   private static FileInputStream configFile;
   private static Properties properties;
   private static String result="";
   private static int ClientCounter=0;
   private static int CPUCounter=0;
   private static int segmentSizeKB=0;
   private static double CPUResult=0.0;
   private static double ClientResult=0.0;
   private static String CPU="";
   private static BufferManager bufferManager;
   private static String tpsResult="",checkpointResult="",cowResult="";
   public static boolean stopPrint=false;
   static DBMSLogger loggerObj = new DBMSLogger();

   //Objects required by CPUUsageThread.IUsageEventListener
   private final int m_PID; // process ID
   private final DecimalFormat m_format;
   private SystemInformation.CPUUsageSnapshot m_prevSnapshot;
   private static double checkpointRes;
   private static double checkpointRes2;
   private static double tpsRes;
   private static double tpsRes2;
   private static double cowRes;
   private static double cowRes2;
```

```
/**
 * This is the constructor for this class. Here all the difrent objects are defined
 * Some parameters is read from the properties file
 *
 * @param segmentSize int
 * @throws Exception
 */
public DataBaseControl() throws Exception
{
    //Load config.properties
    configFile = new FileInputStream("config.properties");
    properties = new Properties();
    properties.load(configFile);
    loggerObj.setLog(java.lang.Integer.parseInt(properties.getProperty("Logging")));
    loggerObj.logMessage("Program started");

    //Initialization of objects used by CPUUsageThread.IUsageEventListener
    m_PID = SystemInformation.getProcessID ();
    m_format = new DecimalFormat ();
    m_format.setMaximumFractionDigits (java.lang.Integer.parseInt
                                       (properties.getProperty("MaximumFractionDigits")));

    //Creates buffermanager, checkpoint and transaction controller objects
    bufferManager = new BufferManager(java.lang.Integer.parseInt(properties.getProperty("DBSizeKB")),
                                      segmentSizeKB, loggerObj,this);

    checkpointObj = new Checkpoint(bufferManager, properties, loggerObj,
                                   this.segmentSizeKB,java.lang.Integer.parseInt
                                   (properties.getProperty("DBSizeKB")),this);

    transactionControllerObj = new TransactionController(
       java.lang.Integer.parseInt(properties.getProperty("DBSizeKB")),
        segmentSizeKB,
         java.lang.Integer.parseInt(properties.getProperty("numberOfTransactionThreads")),
          java.lang.Integer.parseInt(properties.getProperty
                                     ("numberOfOperationsPerTransactionThread")),
          bufferManager,this,loggerObj,(java.lang.Integer.parseInt(properties.getProperty("Load"))));
}
/**
 * This is the main method. Here all the diferent threads are started
 *
 * @param args String[]
 */
public static void main(String args[])
{
    double totalRunTime=0.0;
    try
    {
        segmentSizeKB = java.lang.Integer.parseInt(args[0]);

        // declare a file output object
        FileOutputStream outStream;
```

```
        PrintStream p;

        //Initialize the DataBaseControl Objects
        DataBaseControl DataBaseControl = new DataBaseControl();

       String tempName = java.lang.Integer.toString(segmentSizeKB);
        DateFormat dateFormat = new SimpleDateFormat("hh-mm-ss-dd-MM-yyyy");
        Date thisDate = new Date();
        File resultFile = new File("./Log/" + dateFormat.format(thisDate) +
                                   ".res");
        outStream = new FileOutputStream("./Log/ResultsFormatedForExcel.txt", true);

        p = new PrintStream(outStream);
        BufferedWriter writer = new BufferedWriter(new FileWriter(resultFile));
        String out = new String();

        //Creates a timer thread
        HiResTimer timer = new HiResTimer();
        timer.createTimer();
        timer.startTimer();

        //Execute a new thread which monitors the CPU usage.
        final CPUUsageThread monitor = CPUUsageThread.
                                       getCPUThreadUsageThread();
        monitor.addUsageEventListener(DataBaseControl);
        monitor.start();
        transactionControllerObj.start();
        checkpointObj.start();
        transactionControllerObj.join();
        checkpointObj.terminate();
        checkpointObj.join();
        stopPrint = true;
        loggerObj.logMessage("Starting writing to disk");

        //Writes information to result file
        writer.write(
                "\nDATABASE INFORMATION\n**********************************************\n" +
                     "Database size:\t\t\t " +
                     java.lang.Integer.parseInt(
                     properties.getProperty("DBSizeKB")) + " Kilobyte" +
                     "\n");
        writer.write("Segment size:\t\t\t " + segmentSizeKB + " Kilobyte" +
                     "\n");
        writer.write("Simultaneous threads:\t\t " +
                     java.lang.Integer.
                     parseInt(properties.getProperty("numberOfTransactionThreads")) +
                     "\n");
        writer.write("Operations per thread:\t\t " +
                     java.lang.Integer.
                     parseInt(properties.getProperty("numberOfOperationsPerTransactionThread"))
                     + "\n\nRESULTS\n*************************************"
                     +"****************************" +
                     "\n");
        totalRunTime = timer.endTimer();
        writer.write("The program used:\t\t\t " + totalRunTime +
                     " milliseconds" + "\n");
        writer.write("TPS:\t\t\t\t " +
                     ((java.lang.Integer.parseInt(
                       properties.getProperty("numberOfTransactionThreads")) *
                       java.lang.Integer.parseInt(properties.getProperty(
                "numberOfOperationsPerTransactionThread"))) / (ClientResult)) +
                     "\n");
        writer.write("Number of checkpoints:\t\t " +
                     checkpointObj.CheckpointCounter + "\n");
        writer.write("Number of CopyOnWrite:\t\t " +
                     bufferManager.getCOWCounter() + "\n");
        writer.write("Average COW time:\t " +
                     (bufferManager.getTime() / bufferManager.getCOWCounter()) +
                     "\n");
        writer.write("Average checkpoint time:\t " +
                     (checkpointObj.CheckpointResult /
                      checkpointObj.CheckpointCounter) + " milliseconds" +
                     "\n");
        writer.write("Average CPU usage:\t\t " + (CPUResult / CPUCounter) +
                     " %" + "\n");
        p.println(segmentSizeKB + "\t" +
                  ((java.lang.Integer.parseInt(
                    properties.getProperty("numberOfTransactionThreads")) *
                    java.lang.Integer.parseInt(properties.getProperty(
                "numberOfOperationsPerTransactionThread"))) / (ClientResult)) +
                    "\t" + checkpointObj.CheckpointCounter + "\t" +
                    bufferManager.getCOWCounter() + "\t" +
                    (checkpointObj.CheckpointResult /
                     checkpointObj.CheckpointCounter) + "\t" +
                    ((CPUResult / CPUCounter) / 100) + "\t" +
                    (bufferManager.getTime() / bufferManager.getCOWCounter()) +
                    calculateConfidenceInterval() + "\t" + totalRunTime);
        writer.close();
        p.close();
        loggerObj.logMessage("Program finished");
    }
    catch (Exception ex)
    {
        System.out.println(ex.getMessage());
    }
}
/**
 * Function required by CPUUsageThread.IUsageEventListener
 * @param event CPUUsageSnapshot
 */
public void accept(final SystemInformation.CPUUsageSnapshot event)
{
```

```
    if (m_prevSnapshot != null && !stopPrint)
    {
        CPUResult+=((100.0 *SystemInformation.getProcessCPUUsage(m_prevSnapshot,event)));
        CPUCounter++;
    }
    m_prevSnapshot = event;
}
/**
 * This method makes it posible for the diferent threads to add their result to one commom string
 *
 * @param s String
 * @param res double
 */
public synchronized void addResults(String s,double res)
{
    result+=s;
    ClientResult+=res;
    ClientCounter++;
}
/**
* This method collects results from concerning TPs, checkpoints and COW operations
*
* @param number int
* @param result double
*/
public synchronized void addResultToExcel(int number, double result)
{
    if(number ==0)
    {
        tpsRes+=result;
    }
    else if(number==1)
    {
        checkpointRes+=result;
        checkpointRes2+=result*result;
    }
    else if(number ==2)
    {
        cowRes+=result;
        cowRes2+=result*result;
    }
    else if(number ==3)
    {
        tpsRes2+=result;
    }
}
/**
* This method calculates the confidence interval
*/
public static String calculateConfidenceInterval()
{
    String output="";
    long numThreads=((java.lang.Integer.parseInt(properties.getProperty("numberOfTransactionThreads"))
                    *java.lang.Integer.parseInt(
                            properties.getProperty("numberOfOperationsPerTransactionThread"))));
    double TPS=((java.lang.Integer.parseInt(properties.getProperty("numberOfTransactionThreads"))
                *java.lang.Integer.parseInt(properties.getProperty(
                        "numberOfOperationsPerTransactionThread")))/(ClientResult));
    double stdDeviation=0.0;
    double confidenceInterval=0.0;
    double middelTPS=0.0;

    //Checkpoint
    stdDeviation=Math.sqrt(((checkpointObj.CheckpointCounter*checkpointRes2)-
                            (checkpointRes*checkpointRes))/(checkpointObj.CheckpointCounter
           *(checkpointObj.CheckpointCounter-1)));

    confidenceInterval=1.65*(stdDeviation/Math.sqrt(checkpointObj.CheckpointCounter));
    output+="\t"+ confidenceInterval;

     //COW
     stdDeviation=0.0;
     confidenceInterval=0.0;
     stdDeviation=Math.sqrt(((bufferManager.getCOWCounter()*cowRes2)-(cowRes*cowRes))/
                            (bufferManager.getCOWCounter() *(bufferManager.getCOWCounter()-1)));
     confidenceInterval=1.65*(stdDeviation/Math.sqrt(bufferManager.getCOWCounter()));
     output+="\t"+ confidenceInterval;

     //TPS
     stdDeviation=0.0;
     confidenceInterval=0.0;
     stdDeviation=((numThreads*tpsRes2)-(tpsRes*tpsRes))/(numThreads *(numThreads-1));
     stdDeviation=Math.sqrt(stdDeviation);
     confidenceInterval=1.65*(stdDeviation/Math.sqrt(numThreads));
     output+="\t"+(TPS-confidenceInterval)+"\t"+(TPS+confidenceInterval);
     return output;
   }
}
```

Class BufferManager

java.lang.Object
  └ **BufferManager**

public class **BufferManager**
extends java.lang.Object

Constructor Summary

BufferManager(int databaseSize, int segmentSize, DBMSLogger loggerObj, DataBaseControl dbObj) This is the constructor for BufferManager.	

Method Summary

int	**getCOWCounter**()
double	**getTime**()
void	**lockSegment**(int segment) This method is called from checkpoint and sets the COW lock on the segment This lock is located in the admRow table
private void	**preformCopyOnWrite**(int admRowValue) This is the CopyOnWrite method Data is copied from the locked segment to the segment in the freelist
void	**unLockSegment**(int segment) This method is called from checkpoint and removes the COW lock on the segment
void	**writeToSegment**(int segment, int position, byte value, ClientThread CThread) This method is called from the transaction threads.

```
public class BufferManager
{

   private int freeList[];
   private int numberOfSegmentes=0;
   private int databaseSize=0;
   private int segmentSize=0;
   private int freeSegment;
   private int lockedSegmentPointer=-1;
   private int COWcounter=0;
   private double time;
   private DataBaseControl DataBaseControlObj;
   private Segment[] segmentBuffer;
   private AdmRow[] admRow;
   private HiResTimer timer;
   private DBMSLogger loggerObj;
   /**
    * This is the constructor for BufferManager. Here the memory is allocated and the
    * admRow table is updated with corresponding segment pointers
    * @param databaseSize int This parameter represents the size of the database
    * @param segmentSize int This parameter represents the size of each segment
    * @param loggerObj DBMSLogger
    */
   public BufferManager(int databaseSize, int segmentSize, DBMSLogger loggerObj,DataBaseControl dbObj)
   {
       int i=0;
       timer=new HiResTimer();
       DataBaseControlObj=dbObj;
       timer.createTimer();
       this.loggerObj=loggerObj;
       loggerObj.logMessage("BufferManager.BufferManager("+databaseSize+","+segmentSize+",DBMSLogger)");
       this.databaseSize=databaseSize*1024; //from KB to B
       this.segmentSize=segmentSize*1024;   //from KB to B
       numberOfSegmentes=this.databaseSize / this.segmentSize;
       segmentBuffer=new Segment[numberOfSegmentes+1];
       admRow=new AdmRow[numberOfSegmentes];

       //Allocates the databasememory
       timer.startTimer();

       while(i<numberOfSegmentes)
       {
           segmentBuffer[i] = new Segment(this.segmentSize, loggerObj,i);
           admRow[i]=new AdmRow(i);
           i++;
       }

       //Allocate the free Segment memory
       segmentBuffer[i] = new Segment(this.segmentSize, loggerObj,i);
       freeSegment=i;
       loggerObj.logMessage(numberOfSegmentes+" segments created. Freesegment: "+freeSegment);
   }
   /**
    * This method is called from the transaction threads.
    * If the checkpoint is performing CopyOnWrite is performed
    * The method is calling the setByte() method in segment
    *
    * @param segment int The segment to be updated
    * @param position int The position in the segment
    * @param value byte The value
    */
  public void writeToSegment(int segment,int position, byte value, ClientThread CThread)
   {
         //Performe CopyOnWrite()
         while (admRow[segment].getBusy())
         {
             try
             {
                 loggerObj.logMessage(CThread.getName() + " is sleeping");
                 CThread.sleep(((segmentSize) / (1024 * 1024)) / 2);
             }
             catch (InterruptedException e)
             {
                 loggerObj.logMessage(e.getMessage());
             }
         }

         if(admRow[segment].getcow())
         {
             preformCopyOnWrite(segment);
         }

        //Adds value to given position in the segment
        segmentBuffer[admRow[segment].getsegmentPointer()].setByte(value,position);
   }
/**
    * This method is called from checkpoint and sets the COW lock on the segment
    * This lock is located in the admRow table
    *
    * @param segment int The segment that should be locked
    */
   public void lockSegment(int segment)
   {
       loggerObj.logMessage("Segment"+segment+" is locked");
       lockedSegmentPointer=admRow[segment].getsegmentPointer();
       admRow[segment].setcow(true);
   }
    /**
     * This method is called from checkpoint and removes the COW lock on the
     * segment
     *
     * @param segment int The segment to be unlocked
     */
```

```
  public void unLockSegment(int segment)
  {
      loggerObj.logMessage("Segment"+segment+" is unlocked");
      admRow[segment].setcow(false);
  }
   /**
    * This is the CopyOnWrite method
    * Data is copied from the locked segment to the segment in the freelist
    *
    * @param admRowValue int
    */
  private synchronized void preformCopyOnWrite(int admRowValue)
  {
      admRow[admRowValue].setBusy(true);
      loggerObj.logMessage("setBusy(true)");
      int avSegment=freeSegment;
      double tempTime=0.0;
      loggerObj.logMessage("CopyOnWrite is activated for the"+COWcounter+"th time");
      //Perform the CopyOnWrite operation
      timer.startTimer();

      int lockedSegment=admRow[admRowValue].getsegmentPointer();
      admRow[admRowValue].setsegmentPointer(avSegment);
      admRow[admRowValue].setcow(false);
      loggerObj.logMessage("Copy segment: "+lockedSegment+" to segment "+avSegment);
      loggerObj.logMessage("Free "+segmentBuffer[avSegment].getNumber()
                           + "-Locked "+segmentBuffer[lockedSegment].getNumber());
      segmentBuffer[avSegment].setByteBuffer(segmentBuffer[lockedSegment].getByteBuffer());
      freeSegment=lockedSegment;
      tempTime=timer.endTimer();
      DataBaseControlObj.addResultToExcel(2,tempTime);
      time+=tempTime;

      //Change Segment pointer
      COWcounter++;
      loggerObj.logMessage("setBusy(false)");
      admRow[admRowValue].setBusy(false);
     loggerObj.logMessage("AdmRow table");
     for(int k=0;k<this.numberOfSegmentes;k++)
     {
          loggerObj.logMessage(k+" - "+admRow[k].getsegmentPointer());
     }
      loggerObj.logMessage("Free:"+freeSegment);

  }
  public int getCOWCounter()
  {
      return this.COWcounter;
  }
  public double getTime()
  {
          return this.time;
      }
}
```

Class Segment

java.lang.Object
  └ **Segment**

public class **Segment**
extends java.lang.Object

Constructor Summary

Segment(int segSize, DBMSLogger loggerObj, int number) Constructor for Segment class	

Method Summary

java.nio.ByteBuffer	**getByteBuffer**() This method returns a segment (ByteBuffer)
int	**getNumber**()
void	**setByte**(byte value, int position) This methos sets a byte into the database
void	**setByteBuffer**(java.nio.ByteBuffer buffer) This methos sets a value to a segment (ByteBuffer)

```
import java.nio.ByteBuffer;

public class Segment //extends BufferManager
{
   private ByteBuffer databaseSegment;
   private int segmentSizeB;
   private DBMSLogger loggerObj;
   int segmentNumber=0;
   /**
    *Constructor for Segment class
    *
    * @param segSize int
    * @param loggerObj DBMSLogger
    * @param number int
    */
   public Segment(int segSize, DBMSLogger loggerObj, int number)
   {
       this.segmentSizeB = segSize;
       this.loggerObj = loggerObj;
       this.segmentNumber=number;

       //Allocate direct buffer in memory
       databaseSegment=ByteBuffer.allocateDirect(segmentSizeB);
       loggerObj.logMessage(segmentSizeB+"B of memory allocated");
   }
   /**
    * This method returns a segment (ByteBuffer)
    *
    * @return ByteBuffer
    */
   public ByteBuffer getByteBuffer()
   {
       return this.databaseSegment;
   }
   /**
    * This methos sets a value to a segment (ByteBuffer)
    *
    * @param buffer ByteBuffer
    */
   public void setByteBuffer(ByteBuffer buffer)
   {
     String tempString="";

     try
     {
         this.databaseSegment.clear();//This method resets read / write pointers
         databaseSegment.put(buffer);
     }
     catch (IllegalArgumentException ex)
     {
         System.out.println(ex.toString());
     }
   }
   /**
    *This methos sets a byte into the database
    *
    * @param value byte The byte to be sat into the database
    * @param position int Where to put the byte
    */
   public void setByte(byte value,int position)
   {
       int i=0;
       for(i=position;(i<segmentSizeB)&&(i<(500+position));i++)
       {
           databaseSegment.put(position, value);
       }
       if(i<499)
       {
           loggerObj.logMessage("Segment.setByte(), did not complete 500 updates. Stopped at: "+i);
       }
   }
   public int getNumber()
   {
       return segmentNumber;
   }
}
```

Class AdmRow

```
java.lang.Object
  └ AdmRow
```

public class **AdmRow**
extends java.lang.Object

Constructor Summary

AdmRow(int segmentPointer) This is the constructor	

Method Summary

boolean	**getBusy**()
boolean	**getcow**() This methos returns the value of the COW flag
int	**getsegmentPointer**() This method returns the segmentpointer
void	**setBusy**(boolean i)
void	**setcow**(boolean value) This methos manipulates the COW flag
void	**setsegmentPointer**(int value) This method sets the segmentpointer value

```
public class AdmRow
{
   private int cow;
   private int locked;
   private int dirty;
   private int segmentPointer;
   private boolean busy=false;
   /**
    * This is the constructor
    *
    * @param segmentPointer int
    */
   public AdmRow(int segmentPointer)
   {
       cow=0;
       locked=0;
       dirty=0;
       this.segmentPointer=segmentPointer;
   }
   /**
    * This methos returns the value of the COW flag
    *
    * @return boolean The value of the COW flag
    */
   public synchronized boolean getcow()
   {
       if(cow ==1)
           return true;
       else
           return false;
   }
   /**
    * This methos manipulates the COW flag
    *
    * @param value boolean The value of the COW flag
    */
   public void setcow(boolean value)
   {
       if(value)
           cow=1;
       else
           cow=0;
   }
   /**
    * This method returns the segmentpointer
    *
    * @return int The segmentpointer
    */
   public synchronized int getsegmentPointer()
   {
       return segmentPointer;
   }
   /**
    * This method sets the segmentpointer value
    * @param value int segmentpointer value
    */
   public synchronized void setsegmentPointer(int value)
   {
       segmentPointer=value;
   }
   public void setBusy(boolean i)
   {
       this.busy = i;
   }
   public boolean getBusy()
   {
       return busy;
   }
}
```

Class ClientThread

```
java.lang.Object
  └java.lang.Thread
      └ClientThread
```

All Implemented Interfaces:
java.lang.Runnable

public class **ClientThread**
extends java.lang.Thread

Constructor Summary

ClientThread(DataBaseControl DataBaseControl, BufferManager buffermanager, DBMSLogger logger, int threadNumber, int numberOfThreads, int segSizeKB, int numberSegments, int load, int rounds) This is the constructor for the Client thread.	

Method Summary

void	**run**() This is the main() method of each thread.

```
/**
The Client can perform multiple tasks,  typically 1000 TPS
*/
public class ClientThread extends Thread
{
    private int rounds=0;
    private int threadNumber=0;
    private int numberOfThreads=0;
    private int numberOfSegments;
    private int segmentSizeB;
    private int loadType=0;
    private byte[] byteArray=new byte[10];
    private HiResTimer h = new HiResTimer();
    private String result="";
    private DBMSLogger loggerObj;
    private BufferManager buffermanagerObj;
    private DataBaseControl DataBaseControlObj;
    /**
     * This is the constructor for the Client thread.
     *
     * @param DataBaseControl DataBaseControl
     * @param logger DBMSLogger
     */
  public ClientThread(DataBaseControl DataBaseControl,BufferManager buffermanager, DBMSLogger logger,
                      int threadNumber, int numberOfThreads,int segSizeKB,
                      int numberSegments,int load,int rounds)
   {
       this.segmentSizeB=(segSizeKB*1024);
       this.numberOfSegments=numberSegments;
       this.buffermanagerObj=buffermanager;
       this.DataBaseControlObj=DataBaseControl;
       this.loggerObj=logger;
       this.threadNumber=threadNumber;
       this.numberOfThreads=numberOfThreads;
       this.loadType=load;
       this.rounds=rounds;
   }
  /**
   * This is the main() method of each thread.
   * Here the thread will perform random selected tasks on the Database
   */
   public void run()
   {
       loggerObj.logMessage(this.getName()+".run()");
       result="Results from "+this.getName()+"\n";
       h.createTimer();
       double timeOps=0.0;
       double timeTran=0.0;
       double tpsTran2=0.0;
       double tpsTran=0.0;
       double start=0.0;
       int whichByte=0;
       int whichSegment=0;
       int whichSegmentByte=0;

       //Defines a default byte array to be sure that actual data is put into the database
       byteArray[0]='a';
       byteArray[1]='b';
       byteArray[2]='c';
       byteArray[3]='d';
       byteArray[4]='e';
       byteArray[5]='f';
       byteArray[6]='g';
       byteArray[7]='h';
       byteArray[8]='i';
       byteArray[9]='j';
       timeTran=0.0;

       while(rounds>0)
       {
              timeOps=0.0;
              start=h.startTiming();
              whichByte=(int)(Math.random()*10);

              if(loadType==0)//Uniform load
                  whichSegment=(int)(Math.random()*(numberOfSegments));
              else//Distorted load
                  whichSegment=(int)(Math.random()*(numberOfSegments)*((threadNumber+1)/numberOfThreads));

              whichSegmentByte=(int)(Math.random()*segmentSizeB);
              buffermanagerObj.writeToSegment(whichSegment,whichSegmentByte,byteArray[whichByte],this);

              //The sleep function is removed because this did not give a sufficient load on the system
             try
              {
                  int tall=Math.abs(270/numberOfThreads);
                  if(rounds%(tall)==0)
                  {
                      Thread.sleep(1);
                  }
              }
              catch (InterruptedException e)
              {
                  System.out.println(e.toString());
              }

              timeOps=h.endTiming(start)/1000;
              rounds--;
              timeTran+=timeOps;
              tpsTran=1/timeOps;
              tpsTran2+=((1/timeOps)*(1/timeOps));
       }
       DataBaseControlObj.addResults(result,timeTran);
```

```
        DataBaseControlObj.addResultToExcel(0,tpsTran);
        DataBaseControlObj.addResultToExcel(3,tpsTran2);
        loggerObj.logMessage(this.getName()+" has finished");
    }
}
```

Class TransactionController

```
java.lang.Object
  └java.lang.Thread
      └TransactionController
```

All Implemented Interfaces:

java.lang.Runnable

public class **TransactionController**
extends java.lang.Thread

Constructor Summary

TransactionController(int DBSizeKB, int SegSizeKB, int numberOfThreads, int numberOfOperationsPerThread, BufferManager bufferManager, DataBaseControl dbControl, DBMSLogger loggerObj, int load) This is the constructor.	

Method Summary

void	**createThreads**(int number, int numberOfRoundsPerThread, BufferManager bufferManager) This method creates all the threads and store the referance in the threadArray[] The method also sets name and some referances in the thread object
void	**run**() This is the main method for the transactioncontroller thread

```
public class TransactionController extends Thread
{
   private int numberOfThreads;
   private boolean cont =true;
   private int numberOfOperationsPerThread;
   private BufferManager bufferManager;
   private DataBaseControl DBC;
   private DBMSLogger loggerObj;
   private int segmentSize;
   private int numberOfSegments;
   private int loadType=0;
   ClientThread threadArray[];

   /** This is the constructor. This constructor is called from main(). The method creates
    * several threads wich i stored in the threadArray, then each thread is started.
    * The method waits untill all of the threads have fhinished before it writes out some information
    * <p>
    *
    * @param DBSizeKB                      The size of the database in KB
    * @param SegSizeKB                     The size of each segment in KB
    * @param numberOfThreads               How many thread we shall start
    * @param numberOfOperationsPerThread   How many operations each thead should performe
    * @param bufferManager BufferManager
    * @param dbControl DataBaseControl
    * @param loggerObj DBMSLogger
    * @throws Exception
    *
    */
   public TransactionController(int DBSizeKB, int SegSizeKB, int numberOfThreads,
                                int numberOfOperationsPerThread, BufferManager bufferManager,
                                DataBaseControl dbControl,DBMSLogger loggerObj,int load)throws Exception
   {
       this.segmentSize=SegSizeKB;
       this.numberOfSegments=(DBSizeKB/SegSizeKB);
       this.DBC=dbControl;
       this.bufferManager=bufferManager;
       this.numberOfOperationsPerThread=numberOfOperationsPerThread;
       this.numberOfThreads=numberOfThreads;
       this.loggerObj=loggerObj;
       this.loadType=load;
       loggerObj.logMessage("TrabsactionController("+DBSizeKB+","+SegSizeKB+","+numberOfThreads+","
                            +numberOfOperationsPerThread+",bufferManager,dbControl,loggerObj)");
   }
   /**
    * This is the main method for the transactioncontroller thread
    *
    */
   public void run()
   {
       loggerObj.logMessage("TransactionController.run()");
       //creates a buffer of threads
       threadArray=new ClientThread[numberOfThreads];
       //This method creatres all the threads
       createThreads(numberOfThreads,this.numberOfOperationsPerThread,this.bufferManager);
       int counter =numberOfThreads-1;
       //Wait untill all the threads are fhinished
       while(cont)
       {
           try
           {
               //wait until the thread is finished
               threadArray[counter].join();
               if (counter == 0)
                   cont = false;
               else
                   counter--;
           }
           catch(java.lang.InterruptedException e)
           {
               System.out.println("Exception in TransactionController:" +e.toString());
           }
       }
       loggerObj.logMessage("All clientThreads have finished");
   }
   /** This method creates all the threads and store the referance in the threadArray[]
    * The method also sets name and some referances in the thread object
    * <p>
    *
    * @param number                   Number of threads to be created
    * @param numberOfRoundsPerThread  Defines how many operations (rounds) each thread should perform
    * @param bufferManager            This is a referance to the buffermanager object
    *
    */
   public void createThreads(int number, int numberOfRoundsPerThread,BufferManager bufferManager)
   {
       loggerObj.logMessage("createThreads("+number+","+numberOfRoundsPerThread+",bufferManager)");
       int i =0;
       String name;
       while(i<number)
       {
           name = "Thread";
           ClientThread thread = new ClientThread(DBC,bufferManager,loggerObj,i,number,
                                                  segmentSize,numberOfSegments,this.loadType,
                                                  numberOfRoundsPerThread);

           //creates a dynamic thread name
           name += new Integer(i).toString();

           //sets thread name
           thread.setName(name);

           //starts the thread
           thread.start();
```

```
            //The thread is added to the threadArray[]
            threadArray[i]=thread;
            i++;
        }
        loggerObj.logMessage("All threads have been started");
    }
}
```

Class Checkpoint

```
java.lang.Object
  └java.lang.Thread
      └Checkpoint
```

All Implemented Interfaces:
java.lang.Runnable

public class **Checkpoint**
extends java.lang.Thread

Constructor Summary

Checkpoint(BufferManager bufferManagerObj, java.util.Properties properties, DBMSLogger loggerObj, int segSize, int DBSize, DataBaseControl dbObj) Saves objects that will be used during a checkpoint.	

Method Summary

void	**run**() This is the start() method for Checkpoint The chekcpoint "transfer" one segment at the time until all the segments are "transfered"
void	**terminate**() Terminates the checkpoint thread.
void	**writeSegment**(int position) This method is taking care of the simulation of RDMA The method simulates the transfer of segments from this computer direct into the memory of another computer with the help of RDMA

```
import java.util.Properties;
/**
 *
 * <p>Title: DBMS</p>
 *
 * <p>Description: Frequently takes a checkpoint of the database.</p>
 *
 * <p>Copyright: Copyright (c) 2005</p>
 *
 * <p>Company: </p>
 *
 * @author not attributable
 * @version 1.0
 */
public class Checkpoint extends Thread
{


   private int segmentSizeKB;
   private int numberOfSegments;
   private DBMSLogger loggerObj;
   private DataBaseControl DataBaseControlObj;
   private HiResTimer hTimer = new HiResTimer();
   private long checkpointSleep=0; //Default 1000 ms
   private String ioType = "RDMA";
   private BufferManager bufferManagerObj;
   private IOController io;
   public String CpResult="";
   public double CheckpointResult=0.0;
   public int state=0;
   public int CheckpointCounter=0;
   boolean cont=true;
   /**
    * Saves objects that will be used during a checkpoint.
    * @param bufferManagerObj BufferManager Holds the database
    * @param properties Properties Holds the handler for the properties file
    * @param loggerObj DBMSLogger The object for logging
    */
   public Checkpoint(BufferManager bufferManagerObj, Properties properties, DBMSLogger loggerObj,int segSize,
                     int DBSize,DataBaseControl dbObj)
   {
       this.segmentSizeKB=segSize;
       DataBaseControlObj=dbObj;
       this.numberOfSegments=(DBSize/segSize);
       this.loggerObj=loggerObj;
       this.loggerObj.logMessage("Checkpoint.checkpoint(buffermanager, timerobj,properties, loggerobj)");
       hTimer.createTimer();
       this.bufferManagerObj = bufferManagerObj;
       this.ioType = properties.getProperty("IOtype");
       this.checkpointSleep =java.lang.Long.parseLong(properties.getProperty("checkpointSleep"));
   }
   /**
    * This method is taking care of the simulation of RDMA
    * The method simulates the transfer of segments from this computer direct into the memory of
    * another computer with the help of RDMA
    *
    * @param position int Address of the segment that will be processed.
    */
   public void writeSegment(int position)
   {
       loggerObj.logMessage("Checkpoint.WriteSegment("+position+")");
       bufferManagerObj.lockSegment(position);
       double g=(0.5+ ((double)this.segmentSizeKB/(1024.0*1024.0))*1000.0);
       try
       {
           if( ioType.compareTo("RDMA") == 0 )
               this.io = new RDMA();

           loggerObj.logMessage("RDMA will take "+g+" milliseconds");

           //Define where to write from
           io.write(this.segmentSizeKB);

           //Start the thread
           io.start();

           //Whait for thread to end.
           io.join();
       }
       catch (InterruptedException ex)
       {
           System.out.println("ERROR:" +ex.getMessage());
       }

       bufferManagerObj.unLockSegment(position);
   }
   /**
    * This is the start() method for Checkpoint
    * The chekcpoint "transfer" one segment at the time until all the segments are "transfered"
    */
   public void run()
   {
       double tempTime=0.0;
       loggerObj.logMessage("Checkpoint.run()");
       while (cont)
       {
           try
           {
               hTimer.startTimer();
               for (int i = 0; i < numberOfSegments; i++)
               {
                   writeSegment(i);
```

```
                }
                tempTime=hTimer.endTimer();
                DataBaseControlObj.addResultToExcel(1,tempTime);
                CheckpointResult += tempTime;
                CheckpointCounter++;
                loggerObj.logMessage("Finished the " + (CheckpointCounter) +
                                     "th checkpoint" + " - Time:" + tempTime);
                Thread.sleep(this.checkpointSleep);
            }
            catch (InterruptedException ex)
            {
                System.out.println(ex.toString());
            }
        }
    }
    /**
     * Terminates the checkpoint thread.
     */
    public void terminate()
    {
        cont=false;
    }
}
```

Class IOController

```
java.lang.Object
  └java.lang.Thread
      └IOController
```

All Implemented Interfaces:
java.lang.Runnable

Direct Known Subclasses:
RDMA

public abstract class **IOController**
extends java.lang.Thread

Constructor Summary

IOController()	

Method Summary

abstract void	**write**(double size) Common write method for all IO simulations (network, Disk, etc)

```
public abstract class IOController extends Thread
{
    /**
     * Common write method for all IO simulations (network, Disk, etc)
     * @param size long Size (in kB) of segment that will be processed.
     */
    public abstract void write( double size);
}
```

Class RDMA

```
java.lang.Object
  └java.lang.Thread
      └IOController
          └RDMA
```

All Implemented Interfaces:
java.lang.Runnable

public class **RDMA**
extends IOController

Constructor Summary

RDMA()	

Method Summary

void	**run**() Executed when thread is started.
void	**write**(double size) Simulates the network transfer

```
public class RDMA extends IOController
{
   private double writesleep = 1000.0;
   /**
    * Simulates the network transfer
    * @param size long Size (in kB) of the segment that will be transferred.
    */
   public void write( double size)
   {
       this.writesleep =(0.5+ ((double)size/(1024.0*1024.0))*1000.0);
   }
   /**
    * Executed when thread is started.
    */
   public void run()
   {
      try
      {
              int x=(int)this.writesleep;
              double y=this.writesleep - (double)x;//removing the number before ,
              y=y*1000000;//multiplying to nanoseconds
              Thread.sleep((long)x,(int)y); //this.writesleep);
      }
      catch (InterruptedException ex)
      {
          System.out.println(ex.toString());
      }
  }
}
```

com.vladium.utils

Class CPUUsageThread

```
java.lang.Object
  └java.lang.Thread
      └com.vladium.utils.CPUUsageThread
```

All Implemented Interfaces:

java.lang.Runnable

public class **CPUUsageThread**
extends java.lang.Thread

Constructor Summary

protected	**CPUUsageThread**(long samplingInterval) Protected constructor used by getCPUThreadUsageThread singleton factory method.

Method Summary

void	**addUsageEventListener**(CPUUsageThread.IUsageEventListener listener) Adds a new CPU usage event listener.
static CPUUsageThread	**getCPUThreadUsageThread**() Factory method for obtaining the CPU usage profiling thread singleton.
private void	**notifyListeners**(SystemInformation.CPUUsageSnapshot event) Effects the listener notification.
void	**removeUsageEventListener**(CPUUsageThread.IUsageEventListener listener) Removes a CPU usage event listener [previously added via addUsageEventListener].
void	**run**() Records and broadcasts periodic CPU usage events.
long	**setSamplingInterval**(long samplingInterval) Sets the CPU usage sampling interval.

```
package com.vladium.utils;

import java.util.ArrayList;

// ----------------------------------------------------------------------------
/**
 * This class shows a sample API for recording and reporting periodic CPU usage
 * shapshots. See <code>CPUmon</code> class for a usage example.
 *
 * @author (C) 2002, Vladimir Roubtsov
 */
public class CPUUsageThread extends Thread
{
    /**
     * Any client interested in receiving CPU usage events should implement
     * this interface and call {@link #addUsageEventListener} to add itself
     * as the event listener.
     */
    public static interface IUsageEventListener
    {
        void accept (SystemInformation.CPUUsageSnapshot event);

    } // end of nested interface
    /**
     * Default value for the data sampling interval [in milliseconds]. Currently
     * the value is 500 ms.
     */
    public static final int DEFAULT_SAMPLING_INTERVAL   = 500;
    /**
     * Factory method for obtaining the CPU usage profiling thread singleton.
     * The first call constructs the thread, whose sampling interval will
     * default to {@link #DEFAULT_SAMPLING_INTERVAL} and can be adjusted via
     * {@link #setSamplingInterval}.
     */
    public static synchronized CPUUsageThread getCPUThreadUsageThread ()
    {
        if (s_singleton == null)
        {
            s_singleton = new CPUUsageThread (DEFAULT_SAMPLING_INTERVAL);
        }

        return s_singleton;
    }
    /**
     * Sets the CPU usage sampling interval.
     *
     * @param samplingInterval new sampling interval [in milliseconds].
     * @return previous value of the sampling interval.
     *
     * @throws IllegalArgumentException if 'samplingInterval' is not positive.
     */
    public synchronized long setSamplingInterval (final long samplingInterval)
    {
        if (samplingInterval <= 0)
            throw new IllegalArgumentException ("must be positive: samplingInterval");

        final long old = m_samplingInterval;
        m_samplingInterval = samplingInterval;

        return old;
    }
    /**
     * Adds a new CPU usage event listener. No uniqueness check is performed.
     */
    public synchronized void addUsageEventListener (final IUsageEventListener listener)
    {
        if (listener != null) m_listeners.add (listener);
    }
    /**
     * Removes a CPU usage event listener [previously added via {@link addUsageEventListener}].
     */
    public synchronized void removeUsageEventListener (final IUsageEventListener listener)
    {
        if (listener != null) m_listeners.remove (listener);
    }
    /**
     * Records and broadcasts periodic CPU usage events. Follows the standard interruptible
     * thread termination model.
     */
    public void run ()
    {
        while (! isInterrupted ())
        {
            final SystemInformation.CPUUsageSnapshot snapshot = SystemInformation.makeCPUUsageSnapshot ();
            notifyListeners (snapshot);
            final long sleepTime;
            synchronized (this)
            {
                sleepTime = m_samplingInterval;
            }

            // for simplicity, this assumes that all listeners take a short time to process
            // their accept()s; if that is not the case, you might want to compensate for
            // that by adjusting the value of sleepTime:
            try
            {
                sleep (sleepTime);
            }
            catch (InterruptedException e)
            {
                System.out.println("Exception in CPUUsage: "+e );
                return;
```

```
            }
        }
        // reset the singleton field [Threads are not restartable]:
        synchronized (CPUUsageThread.class)
        {
            s_singleton = null;
        }
    }
    /**
     * Protected constructor used by {@link getCPUThreadUsageThread} singleton
     * factory method. The created thread will be a daemon thread.
     */
    protected CPUUsageThread (final long samplingInterval)
    {
        setName (getClass ().getName () + " [interval: " + samplingInterval + " ms]");
        setDaemon (true);
        setSamplingInterval (samplingInterval);
        m_listeners = new ArrayList ();
    }
    /**
     * Effects the listener notification.
     */
    private void notifyListeners (final SystemInformation.CPUUsageSnapshot event)
    {
        final ArrayList /* <IUsageEventListener> */ listeners;
        synchronized (this)
        {
            listeners = (ArrayList) m_listeners.clone ();
        }

        for (int i = 0; i < listeners.size (); ++ i)
        {
            ((IUsageEventListener) listeners.get (i)).accept (event);
        }
    }
    private long m_samplingInterval; // assertion: non-negative
    private final ArrayList /* <IUsageEventListener> */ m_listeners;
    private static CPUUsageThread s_singleton;
} // end of class
```

Class HiResTimer

```
java.lang.Object
  └ HiResTimer
```

class **HiResTimer**
extends java.lang.Object

Constructor Summary

(package private)	**HiResTimer**()

Method Summary

void	**createTimer**() This method creates a new timer
double	**endTimer**() This method stops the timer
double	**endTiming**(double dStart)
boolean	**isHighResTimerAvailable**()
void	**startTimer**() This method starts the timer
double	**startTiming**()

```
class HiResTimer
{
    public native boolean isHighResTimerAvailable();
    public native double startTiming();
    public native double endTiming( double dStart );
    double dStart=0.0, dEnd=0.0;
    HiResTimer timer;

    static
    {
        System.loadLibrary("hirestimer");
    }
    /**
     * This method creates a new timer
     *
     */
    public void createTimer()//throws Exception
    {
        timer = new HiResTimer();
    }
    /**
     * This method starts the timer
     *
     */
    public void startTimer()
    {
        dStart = timer.startTiming();
    }
    /**
     * This method stops the timer
     *
     * @return double The difference between start and stop
     */
    public double endTimer()
    {
       dEnd = timer.endTiming(dStart);
       return dEnd;
    }
}
```

Appendix III

Sample of result data

SegSize	TPS	∑ Ckp	∑ CoW	Ckp time	CPU Usage	CoW time	C. I. Ckp	C. I. CoW	Lower C.I. boundary	Upper C.I. boundary	Total run-time
100	24140.245908657424	2	506	27566.282324442698	0.555390044255175	0.083723359523592 15	6611.286112248975	7.757293195838202E-4	24030.79070102772	24249.701116287128	55137.06108174231
200	24331.52125740688	3	906	15014.841708553322	0.6158155376860741	0.14960282618628254	1221.078810117171	0.002429839505734875	24221.148452616188	24441.89406219757	45050.73375419978
300	24393.818959088538	4	1460	10459.179254689749	0.6590264131713123	0.21854759823768294	156.05614485948453	0.010236979681896328	24282.73239062136	24504.905527555715	41844.22653558477
400	24421.054857531795	6	1885	7521.693105923238	0.5844149937299579	0.2556681836385164	601.2608318203396	0.0066869441768378	24309.3115775054	24532.79813755819	45138.59918804331
500	24305.110100019563	7	2300	6226.79697016159	0.6530282282656369	0.293507820887234	293.9439747054214	0.00787403794443807	24193.262476940934	24416.957723098192	43596.30768092368
600	24232.942923781025	8	2661	5258.509227716188	0.6941752357298757	0.3332428334625107	90.63981603959077	0.009054201173082917	24121.002186513488	24344.883661048563	42077.58114733881
700	24247.474757878517	10	3089	4396.876178903792	0.6186062550120289	0.3697079354213872	233.17517656071988	0.010009104541327207	24135.28239728224	24359.667118474794	43978.56347661458
800	24136.5386204699	11	3386	3917.1106380690003	0.6456976744186049	0.41711193959791204	126.4468522061339	0.01142276677409721	24024.112388100275	24248.964852839523	43097.089276923754
900	23544.540741507626	12	3856	3643.787708405552	0.6570155351999756	0.45360330972527824	106.57995872552632	0.011563977143407645	23432.1648890503	23656.916593964954	43738.644519958936
1000	23890.761475605203	13	4091	3264.2766829615243	0.7017941727958036	0.5001424953742541	50.859956854724906	0.012668667471764275	23777.7176783499	24003.805272860503	42445.51140705829
1100	23821.91820025833	12	4284	3705.768456071232	0.5765963150260984	0.5462135042340223	188.91924304038636	0.013737601869261328	23709.33331982168	23934.50308069498	44473.95121093542
1200	23606.40776397657	13	4469	3453.960325115994	0.5817998787359107	0.5986282716372523	170.5479701595173	0.014489025476172403	23493.8082739843	23719.00725396884	44910.14380499707
1300	23322.49320731185	14	4792	3246.302685666288	0.5731583118001723	0.6411596355790364	150.5945118713511	0.014999027006936094	23210.038871411092	23434.94754321261	45459.15894152312
1400	23107.14081496376	15	5035	3009.591341328463	0.6112701286571633	0.7046618107544029	121.89130371523461	0.016102483270785145	22994.514308996168	23219.767320931354	45152.33017072423
1500	22995.926235573104	16	5257	2839.6850068621343	0.6110900372807481	0.7283491138160776	112.84950855071651	0.016483814047246883	22883.14537895856	23108.707092187648	45448.11761806037
1600	22444.127622520333	16	5559	2827.9677264712545	0.6549985211784604	0.7784392035159116	101.1637984060979	0.017405434311916223	22331.573519902486	22556.68172513818	45257.17012426004
1700	22537.10923128413	18	5749	2537.4624011904552	0.5980519102129553	0.8326583225238342	83.39712472700651	0.01818503716148972	22424.051772805444	22650.166689762813	45683.26445422911
1800	22292.393233043396	19	6048	2423.0407324942666	0.6276940676543297	0.8773406673271091	75.73594942458747	0.019427762968958213	22178.945628081914	22405.84083800488	46049.83664637886
1900	22104.434499422576	21	6267	2241.1588198470686	0.5770884033827973	0.9252680988868582	76.92672920842848	0.019464388917847034	21990.872651947553	22217.9963468976	47075.32169117647
2000	21935.422036846383	21	6325	2237.507653306109	0.6511265140084024	0.958268319813457	69.59713853684076	0.020243337763697616	21822.55715556126	22048.286918131507	46999.259695482906
2100	21765.46380410194	17	5884	2794.0803139910972	0.6510611654196257	1.0873002826321045	69.99609549707192	0.02362045275390302	21652.579438053952	21878.348170149926	47516.019914604556
2200	21774.626441804536	18	6040	2646.085825126363	0.6270921731965948	1.0676027843084592	74.55892877930843	0.022973546383020477	21661.04478375485	21888.208099854222	47643.852102221215
2300	21489.682406692173	19	6181	2546.783446970547	0.6349549867113069	1.1362794898363646	75.71650179918224	0.023840027021525316	21376.222822394273	21603.141990990072	48403.39992300411
2400	21290.884449457797	19	6307	2502.590817594991	0.6373565476431485	1.1677256887401757	38.62455079527579	0.024579466263659554	21177.87186441922	21403.897034496375	47564.269538957924
2500	21121.820238635995	20	6442	2402.9930591201805	0.6586934597727101	1.2118932733294696	43.23387945275515	0.025123325318852214	21008.720347382983	21234.920129889008	48074.69457128953
2600	20849.44497118151	21	6674	2312.977706173131	0.6523457480576228	1.2680794639409234	42.43042980492706	0.025894211200212975	20736.444506476077	20962.445435886944	48591.05710962082
2700	20703.44353666896	22	6675	2245.5885038252504	0.6597368346694854	1.3110042123096726	46.75870011656629	0.026598779996521042	20590.487300807068	20816.399772530855	49420.921245200254
2800	20439.2767880084	23	6713	2183.1944487453425	0.6481679871325311	1.401854424757915	53.051718836281715	0.028410378859761446	20326.216663142826	20552.336912873972	50229.81792137753
2900	20386.945161320436	24	6922	2093.1189328374708	0.6426229442312038	1.4075182515836473	47.829947007427	0.028355057370848973	20273.824018253577	20500.066304387296	50254.020768225695
3000	20043.116571377665	25	6951	2055.39000701296	0.6577185484263696	1.516336936973679	58.517634728595745	0.030067789281031546	19929.194973298363	20157.038169456966	51401.74576355928
3100	20091.70639216082	26	6939	1979.3709962391745	0.6590234360369174	1.5039352410980422	60.07996344822111	0.03006940033479767	19978.571753512624	20204.841030809017	51479.059662484666
3200	20156.91199563062	23	6551	2190.9201886334176	0.6795506946639575	1.565991626836437	43.377131064378126	0.03225141081476131	20043.736426734402	20270.08756452684	50405.80837022026
3300	20239.169106663117	24	6482	2118.0542704389104	0.6721618719108305	1.5911879819561925	50.89772135164559	0.032727319844032356	20125.91462011149	20352.423593214746	50851.048594074986
3400	20055.766084977087	25	6547	2061.9432323876063	0.6518881200406704	1.6500927784131565	51.09673024004905	0.033856225539707165	19942.541204134915	20168.99096581926	51572.39765979148
3500	19719.941139400154	26	6702	2011.2891764490507	0.6768160616953215	1.7299518535277294	53.76355906723567	0.03507382791215198	19606.80744782015	19833.07483098016	52308.785293117704
3600	19697.337304520763	26	6749	1971.384596616386	0.6704892929400255	1.733469867876607	39.94277377216853	0.03500539393143691	19584.022677063636	19810.65193197789	51272.71122540131
3700	19168.845964077336	28	6856	1927.5048283972471	0.6718448388523954	1.9182359523964312	55.00447235619088	0.03844285528476031	19055.65903155591	19282.03289659876	53986.96362294011
3800	19359.709671632638	28	6947	1875.0175762649799	0.6894614390372739	1.8191855941091384	42.729036132070874	0.03633319668702734	19246.415320157048	19473.004023108228	52516.20641799104
3900	19104.04533649317	29	6902	1852.344428412094	0.6841246588091505	1.918012335729619	52.92035455013195	0.03838800696161546	18990.979521583737	19217.111151402605	53736.71495186923
4000	19087.080225418034	30	6938	1794.3396085319894	0.6718305318323501	1.9289197214757239	47.90821884049571	0.03845532145610525	18973.608809984693	19200.551640851376	53852.95111527385
4100	18954.777415713357	31	7030	1749.2955048849008	0.6677629755833324	1.9674007635192625	50.41544869285729	0.03912168290603398	18840.81843060412	19068.736400822596	54238.233794197644
4200	19161.385278113787	28	6469	1883.6873669832844	0.6922104742637639	2.0480029376018662	54.274051642322014	0.042211906423967216	19047.44320628761	19275.327349939962	52763.10419044318
4300	19089.271530004764	29	6545	1825.8255105244957	0.6522744193214277	2.0510037380130166	49.18997261843283	0.04182008960061885	18975.227731157396	19203.31532885213	52962.61959695483
4400	18578.20003296719	30	6602	1827.260346948163	0.6571656313027222	2.259427427628982	67.72232906857144	0.04590933839014754	18464.178611513085	18692.221454421295	54836.04723581409
4500	18794.019627928545	31	6711	1741.5074153464502	0.6543559384165811	2.141680270173005	53.74306568919037	0.04341297221910854	18679.969030017266	18908.070225839823	54004.85480107728
4600	18414.59906494114	32	6617	1744.4364102768686	0.6445383009080037	2.325353136221228	60.161316233902504	0.04700325294025311	18300.526793744506	18528.671336137773	55836.89849441363
4700	18501.817479340927	33	6761	1674.7002461585885	0.6637846463685038	2.236672176478151	48.308989389699846	0.04487720774960533	18387.67524394636	18615.959714735494	55280.74040384513
4800	18322.339035874513	33	6706	1673.1188263292624	0.6477477507064319	2.33178307282206	60.70324727511641	0.046947997111305206	18208.299541838456	18436.37852991057	55233.25097428137
4900	18221.53829427237	34	6715	1648.806489153912	0.6657735075332728	2.3730795038660086	57.56749280199217	0.04776812373526327	18107.570388120712	18335.506200424028	56073.558985187454
5000	18249.879332928773	34	6685	1630.053068963773	0.6578600850885516	2.3650844263125195	56.30156476502162	0.048036151767337686	18136.05095391284	18363.707711944706	55436.202750186654

5100	17598.79037381552	35	6605	1666.3320216350364	0.6554247850714431	2.608609301927316	83.605027158669	0.05345421700603164	17484.78586056424	17712.794887066797	58347.242574729346
5200	18123.224405117468	36	6688	1567.9204291974804	0.6575145299183264	2.428981350011827	54.4948647624788	0.04912547027287377	18009.171290967162	18237.277519267773	56461.7862073548
5300	18144.463234216O2	32	5929	1750.2377091450953	0.6541679438898532	2.721401699133759S	58.269391921684S	0.05850450665040243	18030.58525312443	18258.341215307606	56025.2491893765T
5400	18600.86975984571	32	5856	1703.2040664497893	0.6443917418251314	2.515093648784996G	52.829968094552655	0.05455611925734201	18486.6649907326T	18715.074528958754	54521.8848211428T
5500	18106.986323398134	33	5885	1701.4362065841337	0.6498712281272374	2.729948944958285	54.636555030301245	0.05834970815600286A	17993.736897237A9	18220.23574955878	56166.10403211829
5600	18190.19049591996	34	5934	1649.882044457629	0.6421919670856225	2.683104167710322	57.558535263657S5	0.05720488222081003	18076.052567797276	18304.328424042644	56116.8050587302
5700	18182.887024630545	34	5912	1634.2737265385058	0.6479238943816423	2.698676849830521	53.144268640857S	0.05836912620319713	18069.278247767998	18296.495801493093	55589.38345054931
5800	17780.38772266146	35	5924	1637.7520752321777	0.6553453568594927	2.903366594924981	59.745896345216676	0.06201130072158303S	17666.1653540014S	17894.61009132144	57348.278037504664
5900	17957.851150338513	36	5958	1580.1092604875892	0.6313941739580443	2.744771924740933Z	53.419082139462155	0.05856410647070311	17843.870116525708	18071.832184151317	56901.4452712522
6000	17520.95299208669	36	5851	1619.2692344834197	0.6763127012921626	3.081886681306021	70.98561417043582	0.06595668003975029	17407.408430852272	17634.49755332111	58318.5694662951S
6100	18015.535549626642	37	5920	1533.1223534834899	0.6642467218489131	2.782975345368970T	55.207095830589815	0.05969588220496774	17902.021829581856	18129.049269671A3	56747.715332848915
6200	17383.701077708796	37	5922	1573.65536143478	0.6781512159928164	3.140713777711003	65.715091501009O8	0.06655820733658525	17269.556891240038	17497.845264177555	58246.9682200282G
6300	18219.215733304296	34	5331	1636.5599323513405	0.6637089130181927	2.973509917650259T	56.164771501445911	0.06745346576044008	18105.859169747911	18332.572296860682	55664.042728721135
6400	18070.148870607165	34	5282	1647.330519205642	0.6817913465508859	3.085882198424622S	58.252689504810S8	0.07047039745303563	17956.792611844S6	18183.505129369T7	56028.482131952966
6500	17710.44292987103	36	5489	1584.4905106171211	0.6854332774198089	3.173647362904004	45.566600393737877	0.07116362002489385	17596.701655628804	17824.18420411326	57060.1665834488
6600	17998.348931653785	36	5385	1595.5919706319369	0.6703783032989493	3.085311400909745T	53.086734445019S	0.06951339861782788	17884.983608364A7	18111.7142549431	56422.1046887499T
6700	17231.014434345583	38	5581	1555.180331434955	0.6875983411847958	3.436368139622875	57.706310920294S6	0.07557081554086227	17116.901958977895	17345.126909713277	59122.967695056264
6800	17812.87290235077	37	5375	1545.9845987493277	0.6761523854486682	3.121342641632762S	51.219119023901955	0.07023539581171281	17699.538227186786	17926.207577514753	57224.9939829875T
6900	17264.074209781957	39	5483	1525.299023445245	0.6718400743430882	3.499423049611681T7	63.048157839915720	0.07851896408964754	17150.062035457T3	17378.086384106184	59506.48676837236
7000	17479.865392694162	40	5590	1458.6213236877368	0.6826526337395358	3.289813672907031S	48.903704139934476	0.07223219805374677	17365.530994946083	17594.199790442242	58363.1576825823
7100	17168.0023095224T	40	5563	1474.3890782158285	0.6843321490898824	3.489866056805088T	57.630467226434372	0.07748234642022282	17053.720324022564	17282.28429502238	58997.7771068876
7200	17233.613933744211	40	5479	1470.5539481560984	0.673184963625955	3.498194507742699T	58.649194299466235	0.07829301810845848	17119.825436904714	17347.402430579503	58842.7258389552G
7300	17135.81169814357	42	5580	1427.337049476409	0.6704073433633861	3.515676061673375S	52.389138847857S3	0.07659402897825G4	17021.643959886173	17249.979436400971	59972.0994302971
7400	17304.241223824327	38	5085	1542.830783805566	0.6706169996876872	3.728072011773922S	60.082616333116O4	0.08623862275823278	17190.230312915242	17418.25213473341	58644.852397205T8
7500	17833.886500031556	38	5005	1499.291084940382T	0.6470648436601061	3.435363816227505T	55.508146371992716	0.08011335574814501	17719.833187342596	17947.939812720517	56992.2622853447T
7600	16994.364953666027	39	5079	1531.338016912602T	0.6608985680960153	3.983669406962859T	60.753632334516155	0.09118955235124875	16880.13533796027	17108.594569371784	59740.1406691643T
7700	17553.855980120614	40	5115	1446.011792874T8	0.6743433339433024T	3.547298234891524	49.402993742403716	0.08194143369585914	17440.003953617193	17667.708006624034	57858.916768772T3
7800	16975.29803425274	41	5150	1469.597387919798G	0.6557132181786798	3.904730976663756T	60.163316538739345	0.08983465087409978	16860.998840251574	17089.597228253908	60275.079051450586
7900	17078.277358916323	42	5182	1427.458787626564T	0.6550440262779358	3.835273623178539T	53.56878531797464	0.08691240323854049	16963.818763923784	17192.735953908T6	59974.2541331129
8000	17236.549902171908	40	4977	1466.728422734121S	0.6702350428525249	3.855273551574945	59.615037816897654	0.09055014219506773	17122.564729225942	17350.533075117874	58699.3197546130T
8100	16820.7328906977	42	5160	1429.125780847244	0.6774865869862396	4.013073413797763	61.759906687825T1	0.09285534955233453	16706.504471896T8	16934.961309498725	60040.8361897365S
8200	17129.094779208062	43	5131	1383.977579736O315	0.6651904996983156	3.810505996361808G	53.299666361457O3	0.0876743143816T9	17014.851290284176	17243.338268131T95	59529.208706535G5
8300	16300.540297984284	43	5142	1442.159401762139	0.6725640231760974	4.300293162375827	64.358634485725T	0.09774949079745868	16186.182502453115	16414.898093515454	62032.336901031944
8400	17799.906122393895	38	4479	1502.389149958624	0.6696557350953856	3.832532515286020G	47.102354987178S5	0.09436375423960S2	17686.433332809753	17913.378911978038	57112.67849814676T
8500	17018.88363657476	39	4537	1527.456290903801S	0.6794522582359574	4.394380637587806S	57.427377157795796	0.10637603526446T6	16905.159850071S6	17132.607423077T96	59593.8499813343
8600	17352.97766116484	40	4613	1459.852885196123Z	0.6737704236189371	4.071523842740222S	43.159264037315815	0.09921307921960S9	17238.895810496975	17467.059511832707	58417.2180500373
8700	16933.869300315681	41	4671	1452.699705479122	0.6712603475041S7	4.336509892336T3	46.300965122826T1	0.10310351351855T3	16819.637156932535	17048.101449381087	59583.195571902834
8800	17157.214893099826	40	4490	1477.227680140392T	0.6763879896145T8	4.317664505109084	55.669135475840109	0.10626171245621T8	17043.249185732533	17271.180600467T2	59109.41897265479
8900	17197.499887462425	41	4568	1437.272082521452	0.6889899201892T6	4.252054873404408	46.892897993838886	0.10264151248671105	17083.443912580133	17311.555862344718	58955.7899120047
9000	16703.927999729293	41	4551	1477.932624394656	0.6868377345655624	4.671055932975315	59.903329302871139	0.11235294277045528	16590.055375111107	16817.800624474752	60619.055148725T
9100	17442.2615742841	41	4409	1426.335869046334	0.6773518424482728	4.184271467941295	50.882008657549896	0.10371974869117435	17328.663680987425	17555.859467580776	58505.3643815663T6
9200	16534.306078350235	42	4497	1460.903076177207T	0.6889668225124894	4.755088397657419	60.640667556905T5	0.11687870142356S9	16420.01593561431	16648.596221086T5	61390.1464538557T9
9300	17028.640273660876	43	4575	1384.858156913801S	0.6796977611132252	4.367424886337578	47.050957604309T6	0.10566791425520716	16914.537445870646	17142.743101451106	59576.427465354T
9400	16526.709704046413	44	4567	1408.045398697525	0.6807108004499121	4.758354002084797	58.407587694647226	0.11450270708055T1	16412.605963185022	16640.81344494324	61981.0160481441
9500	17006.351962539473	42	4304	1456.837432059178T	0.6554531327130512	4.704842574520554	48.446864724756374	0.11655893762624475	16892.521837997O1	17120.182087081936	61205.0688069696
9600	17038.310223813573	42	4330	1422.565259352834T	0.6662813639230021	4.640890453125023	45.953496527074T2	0.11628874426665352	16923.981139631T7	17152.639307995374	59772.991694776276
9700	16675.045844586308	43	4371	1429.700934973903	0.6757256532612402	4.882446620386771	49.996295550044S9	0.12125949252507691	16560.725853671225	16789.365835501T9	61499.432705252T7
9800	17111.970613993584	42	4231	1422.918332984574	0.6535319346312933	4.550491768383555	49.696041748302T8	0.11498956651653945	16997.892872047T7	17226.048355939398	59781.912366340T6
9900	16589.883965704576	42	4240	1447.871093176618Z	0.6734986167539757	5.107478562699931S	61.415052689845S4	0.13016529164160412	16475.763209743818	16704.004721665333	60835.967355074T
10000	16878.428156745674	44	4307	1391.237616690746T	0.6574052020389761	4.727366167976659	45.700490799994O1	0.12990937991173673	16764.061661068O7	16992.794652423272	61254.216658444886
10100	16428.270131521305	44	4284	1420.441302016922S	0.6686313104873I	5.188934311458563S	54.643424979477S6	0.12949145110134866	16314.680073699672	16541.86018934294	62529.19139479228

10200	16669.051710956446	45	4342	1368.76767297596	0.6622737769819639	4.854261528668119	43.894202924684194	0.12142930614362601	16554.7614145292	16783.342007359974	61617.3834945469
10300	16413.356067429533	46	4433	1344.9787108890428	0.6744774733898329	5.005913801824103	44.193141928338996	0.12383177615393039	16298.890317128038	16527.821817731026	61890.07089121913
10400	16427.419241400832	45	4304	1374.4416124325223	0.6732496183634977	5.200102844499461	54.26270543219791	0.12913018031646345	16313.315467118911	16541.523015682753	61874.523640405845
10500	17165.91440459293	42	3927	1415.7390324836606	0.6455996360696714	4.963135373718856	47.871160468391246	0.12940835433174763	17051.386773417	17280.44203576886	59483.2796684176
10600	16651.756708570596	43	3969	1428.8041708938313	0.6598658108016722	5.399591388383696	51.56472822073112	0.14132918750729612	16537.271595153183	16766.24182198801	61472.07183950189
10700	17268.934277347445	43	3945	1379.1161957253755	0.6294672918514798	4.857326309052275	44.10025935172886	0.12715623203590584	17155.144124250037	17382.724430448454	59320.372051823906
10800	16202.115444890824	45	4106	1398.2194461842950	0.6727613531161067	5.640140799569759	51.70790125443319	0.14419546065818128	16087.722354634116	16316.508535147532	62942.64251673244
10900	16754.719706490927	45	4080	1343.1841465195903	0.6629055156397 94	5.169685963153864 5	40.99950558506972	0.13170847575445038	16640.258454257186	16869.180958724668	60467.240810490104
11000	16607.323522095845	44	3957	1380.0621729775339	0.6663177573053334	5.414536569000717	49.986332529501 87	0.14259968899157294	16493.200309561453	16721.446734630237	60746.24988267292
11100	16576.520398806737	45	4006	1364.3109094107574	0.6640377824273496	5.41555398215342	48.700878278063435	0.14037115567738498	16462.074739289004	16690.966060324 47	61420.111415724496
11200	16564.643865999682	46	4055	1332.9887867871808	0.6531404928530 93	5.325940688342058	45.438780329200 03	0.13732505286044386	16450.238667993 97	16679.049064005394	61337.925396645514
11300	16227.379718888287	46	3964	1373.3511527501141	0.6757902759166258	5.800140294233055 5	55.64064214413136	0.14946506485700006	16113.381638822291	16341.377798953665	63201.772736454055
11400	16839.213166336136	46	3921	1324.7892536093723	0.6512186127278886	5.292738519983181	49.729234845129 34	0.13732012565282 92	16724.732437854 61	16953.693894817 66	60967.6400148658 7
11500	15893.239911044822	48	4084	1332.566736412947	0.6917889559921513	5.927633403952715	52.970312012937 62	0.15206920797572504	15779.414699243089	16007.065122846556	63989.4625136659 4
11600	17161.343236639597	43	3636	1387.1326244150698	0.6502009457440877	5.352317786817411	43.371851817458 75	0.14623373548412968	17047.494866837656	17275.191606441538	59672.880546370 86
11700	16517.992845115674	44	3701	1398.4006755018 52	0.6562546961346416	5.875593988765505	47.739005353450366	0.15821156265071307	16403.822614235 81	16632.163075995537	61556.61731274332
11800	16655.643808174 47	44	3670	1382.4187213878897	0.6590915348190333	5.663036718423174	46.081695774695504	0.15425361021087206	16541.46718719 64	16769.820429152544	60853.491800437 31
11900	16373.164583367852	45	3721	1385.9818471533372	0.6605313367176747	5.780974955451315	46.610531535331795	0.15642558251891264	16258.991838388032	16487.337328347672	62409.6132299610 7
12000	16302.746665720924	46	3766	1358.5638124444956	0.6655063292522159	6.001353119371958	45.952417815450 69	0.15986019543980 5	16188.515518708515	16416.977812733334	62516.975803957124
12100	16956.096067689563	44	3600	1353.7914966504813	0.6634742094962887	5.594910964007859	47.154256338283524	0.15414181648948683	16842.183944733602	17070.008190645523	59592.2836898 6987
12200	15741.7506940441	48	3846	1345.5120265959986	0.6794616078810 84	6.314333698354609	45.982703692980 65	0.16688445176070923	15627.303980741939	15856.197407346 26	64610.386527718525
12300	16797.49253283274	46	3695	1313.6253010129312	0.6567639171119639	5.579155361601284	41.634864781132 86	0.15092565147440917	16683.329414897 09	16911.655650768393	60455.216844434966
12400	15939.854515193863	48	3796	1328.7592003426482	0.6785329477733812	6.325286726629218	46.77626728123639	0.16828612008951 4	15825.376327524 97	16054.133702862757	63809.3560243453 7
12500	16308.07183996497	48	3773	1306.9182000571 08	0.6745031376436914	5.961586636971595	43.069545935522 05	0.15798332605314966	16193.963148428153	16422.180531501 79	62758.170794890 94
12600	16416.925240714143	46	3562	1357.2169243791 62	0.6749770389446386	6.239137181352937	44.128734230644795	0.16966507478865506	16302.410462973317	16531.440018454967	62453.527225214 65
12700	16446.32174979176	46	3567	1339.2831551085392	0.6740348491192423	6.195898500309141	43.300588493543 91	0.17081983214988442	16331.748857128965	16560.894642454557	61631.797465801 82
12800	16608.233872372 06	46	3562	1327.7392995489513	0.6785329477733812	6.027759441356831	40.306397709986 82	0.16607207162887533	16493.750370541256	16722.717374202864	61100.237655658 36
12900	15990.806924474211	48	3659	1323.2228018325131	0.6639134385989032	6.521818853272356	42.407080659075 42	0.17672075768004322	15876.241818018 2	16105.372030866602	63542.160677097 22
13000	16741.318821085384	47	3548	1291.4002136056286	0.6750000677353802	5.930194905482526 5	38.265837376797 42	0.16235572303609527	16626.764258775656	16855.873383395112	60721.824072716125
13100	15883.879929250115	48	3600	1335.5415989411 12	0.6424935201004596	6.729719776889 67	48.399491565513 65	0.18323078547904 79	15769.262012578525	15998.497845921705	64126.9208475548
13200	16505.84441149052	48	3542	1288.5638671631957	0.6608165563810953	6.160638459907041	41.415599668154 35	0.16858312502570108	16391.382303469738	16620.306519511305	61878.027431203 67
13300	16039.213037591153	48	3580	1314.8642900573 08	0.6559883940693908	6.594522556562345	46.518853823726026	0.18138357491868726	15924.868841288842	16153.557233893463	63133.935776758575
13400	16031.094750911461	49	3596	1295.2367610870276	0.6679146740455 52	6.503392647479 29	46.240197847 09921	0.17965613264 19214	15916.631067176604	16145.558434646318	63497.273868727
13500	16023.903282301944	50	3653	1270.6089531091673	0.6708341570290 09	6.460572728848565	39.304750289272334	0.17473522392644836	15909.518299168521	16138.288265435367	63561.61026011946
13600	15812.255139553572	50	3604	1287.1002547330813	0.6422598851320838	6.762028831547767	44.474045404551475	0.18410596731669 26	15698.033241961 91	15926.477037145234	64381.416122140 15
13700	17042.6864768051	44	3115	1361.9285649007813	0.6294984748471528	6.318772263981129	52.376021792432 47	0.18437169276066	16928.226020561877	17157.146933048 32	59951.771314863 21
13800	16173.438958220804	46	3298	1360.3025751382117 05	0.6522375132618105	6.904667421376093	49.784876719635 03	0.19812299100866365	16058.794031792626	16288.083884648982	62603.499594354966
13900	17111.492769473185	44	3108	1344.5885441185562	0.6320977880708777	6.266137415030701	50.088721377275 31	0.18497050861611095	16996.933098544367	17226.052440402003	59187.702477534534
14000	16217.646735762612	46	3259	1353.3358101915721	0.6638555912495498	6.926170349057287	51.254488664061576	0.20029858456241 93	16103.022559301298	16332.270912223925	62279.844680617 03
14100	16674.080965794 34	46	3219	1315.1628598286347	0.6363393052308868	6.545439105109599	47.092924325887 84	0.19080337088646 97	16559.413123958362	16788.748807630316	60522.546973494 75
14200	16405.887192403028	47	3245	1324.6969166538067	0.6444023079118952	6.807912257211 72	49.415428980990185	0.19627148824813 56	16291.244866921 43	16520.529515884627	62287.783134899 47
14300	16191.086252279341	48	3297	1301.6216640945818	0.6392818046744696	6.900417339181591	46.256836721745 95	0.19782916901685024	16076.464416767 35	16305.708087791332	62503.925289984 53
14400	16520.528551476797	47	3231	1304.1981060939 16	0.6357025751783933	6.708354837293219	46.532590757751 83	0.19193440806074652	16405.92416480319	16635.132938150404	61321.820697896 11
14500	15883.024610507799	48	3247	1328.5232418715445	0.7049133703906793	7.302118771856035	55.171434760484 62	0.21069843953106 83	15768.776829532 86	15997.272391482737	63796.104171110 88
14600	16470.222506666047	49	3324	1257.3504353849428	0.6476525434080 15	6.625361196139343	37.902626827154 77	0.18702937508449 11	16356.191847675856	16584.253165656235	61632.222480467 71
14700	16000.100766209796	46	3099	1381.9786965999494	0.6121967858267557	7.491432833165 97	55.380685388534 54	0.22043459390875098	15885.533088731394	16114.668443688199	63597.374167711 05
14800	16922.488100369 07	44	2925	1362.599253123167	0.6253946341688631	6.782185546343503	56.314592962786 01	0.20579537473587317	16807.936097789323	17037.040102948 82	59978.467988373 95
14900	16145.889664903665	46	3020	1380.1905476200 69	0.6320571985814076	7.425229372203918	58.432514763926164	0.22552758801389108	16031.29551769869	16260.483812108 64	63513.911771372 19
15000	16361.752746534485	46	2988	1357.2906598314758	0.6255192681738 25	7.263260096900154	56.182645004431 11	0.21653809653461958	16247.501967601529	16476.003525467 44	62460.364420564234
15100	16745.969527315177	44	2898	1371.9275139871331	0.6291603534783373	7.040287610701706	62.138507741463 57	0.21588097367927475	16631.277891610316	16860.661163020 04	60392.703805797 03
15200	15932.61426735854	48	3058	1341.210528126	0.6109813583780234	7.704728668231648	56.338765049576935	0.22567134285876 78	15818.008499387779	16047.220035329301	64404.846712175 35

15300	16998.97416632353	44	2859	1348.661239105126 5	0.6182576509245865	6.832513969780765 5	58.537424333206358	0.2113573722493721	16884.41209654365	17113.536235721058	59365.560496773505
15400	15790.047410887939	48	3067	1331.739939390684	0.6234668382801844	7.870332548572073	56.82640251994045	0.2346325404981356	15675.475333485412	15904.619488290466	63946.615968815
15500	16600.553058782567	47	2986	1303.582550526383	0.6347374352629503	7.062336651042816	52.77605784692305	0.2108258215193011	16485.90953378531	16715.196583779823	61296.739695549457
15600	15837.762878256637	48	3058	1330.228006714642 3	0.6222334879442257	7.770477346941761	57.2730825096271	0.2311660677443282 11	15723.170169694004	15952.355586819277	63876.243774665 35
15700	16094.600347388005	48	2985	1316.923853790298 4	0.6176698793510993	7.498784391353974	56.21671344649969	0.2240030408702232	15980.070013650595	16209.130681125414	63236.627214881875
15800	16402.191038534475	46	2895	1338.235128119254 1	0.6169788848883745	7.538937776283487	53.52063095120479 4	0.2312885231401646	16287.57364402303	16516.80843304592	61580.3945422910 8
15900	16046.179568673073	48	2962	1313.187540949205 2	0.6228600735950223	7.826754328110373	50.2718845790195 3	0.2367310496905814	15931.484105295185	16160.875032050 96	63053.485867420 4
16000	16598.912758204366	46	2840	1325.483181599880 3	0.6065019390124873	7.362922108238725	53.68702602723729 5	0.2276228331210558	16484.610899371954	16713.214617036778	61003.921057543 6
16100	16075.045846496383	46	2848	1365.276090550533	0.6362972723786 09	8.109928844408 25	62.44587308958245 6	0.2509208420143763	15960.534182241212	16189.557510751554	62827.709215081 9
16200	16722.059881363 9	47	2844	1296.835351965625 7	0.6164852781987322	7.304856514309867	51.91651448122233 6	0.2239266694036489	16607.374660128 02	16836.745102599 78	60977.934673484 08
16300	15866.567191283635	48	2906	1330.985170582568 8	0.6371921527716217	8.243237602345 08	57.7032733263587 2	0.2511390413534030 4	15751.952898396246	15981.181484171024	63910.427693856 33
16400	16049.142565318532	50	2987	1260.538266785771 6	0.6409180018810 07	7.656475083185835 5	43.53859546913943	0.2307548655361757	15934.485362169 48	16163.799768467585	63051.757612260 68
16500	15658.132121115725	50	2960	1300.944119460295 3	0.6425956013450483	8.173586526398749	52.88847600444915	0.2455788542271212	15544.190988681079	15772.073253550 37	65078.470993213 7
16600	15568.887118889455	52	3072	1260.631991780951 2	0.6513415899867502	8.113532859519204	43.39265386373 64	0.2383903297236735	15454.619117627328	15683.155120151583	65579.570603508 13
16700	15844.893813492097	52	3028	1237.002310947262 5	0.6363849013620356	7.865082210146516	42.69535336393755 6	0.2332468785286883	15730.21702328529	15959.570603698905	64354.160695429 58
16800	15889.09961071555	48	2786	1387.978197155874	0.6242926910563558	8.514483710426 1	52.27068110376545	0.2622389514298004	15774.495203647846	16003.704017783253	64257.113981254 33
16900	16693.964896095924	46	2685	1324.077124002662	0.6280973520975947	7.667723715566976	48.45589910667785 5	0.2417830139935616	16579.460556510625	16808.469235681223	60940.769135646 61
17000	15758.668609723645	49	2824	1318.773926669489	0.6415471195362683	8.751900011154298	51.10177705252839	0.2668571015450378	15644.062034982331	15873.275184464 96	64644.190019865 61
17100	16479.546719542053	47	2725	1305.589522012727	0.6345123583269745	7.936522895292666	48.89936035552712	0.2461834164356026	16364.865731615058	16594.227707469046	61391.256419657 62
17200	15714.557953325684	50	2845	1303.509621186 87	0.6425002363377036	8.589367151853839	47.07432370476956	0.2610794021603023 4	15600.323064169 74	15828.792842481394	65200.367270788 2
17300	16231.341026511738	48	2688	1311.161005058412	0.6236742619529991	8.321865308461 52	53.98532431098852	0.2606935861849307	16116.655944483387	16346.026108540089	62967.034043184365
17400	16199.578773531217	48	2734	1298.424156954685	0.6337479564060406	8.212525979107038	51.14324012471441	0.2594851667847546 4	16085.044368654 7	16314.113160196965	62355.534134179514
17500	15939.97171963546	48	2713	1325.927321362304 8	0.6402687027488538	8.671272966314136	58.12049412226307 5	0.2728897841302856	15825.345012872267	16054.598426398652	63668.729139445895
17600	16215.700427853766	50	2749	1262.052563216628 4	0.6205767139341672	8.123444623756 13	46.78282424697394 5	0.2514399249833645 4	16101.34594819282	16330.054907514712	63126.211843701 67
17700	15491.81679348841	51	2825	1291.841569364666 4	0.6545192614586387	9.089160411354647	52.65366561215371 6	0.2802813548279518	15377.166914584866	15606.466672391953	65911.397255479 71
17800	16419.63048529254	49	2715	1268.042197797070 2	0.6256514611285 95	8.052476043389404	48.19503115687750 5	0.2530230180069773	16304.897167838873	16534.363802746204	62160.344481627646
17900	15933.543165035084	49	2644	1311.553078231765	0.6298086757562758	8.935806880768682	50.79751858789923	0.2866235660249374	15818.804487943042	16048.281842127126	64295.210717095 09
18000	16071.562096245003	49	2680	1290.685307423059 6	0.6206160523681782	8.523990049510795	41.30430776441929	0.2688276776850535	15956.920709629578	16186.203482860428	63271.580738360 62
18100	16086.964984937826	49	2654	1297.098735550 43	0.6294780339916242	8.718056346716017	46.62859480420332 4	0.2781189266686169	15972.252237348031	16201.677732527 62	63586.571503853 13
18200	15869.040738740467	50	2671	1280.891154318436 3	0.6314958862442694	8.929260653833383	47.10299909186175	0.2824295120016941	15754.303531976715	15983.777945504218	64069.330756692 98
18300	16127.737810676106	50	2655	1267.671763439228 3	0.6350874109745189	8.588406080991446	44.04041717189065	0.2706180013740747 7	16013.047378564337	16242.428242787875	63411.889952535865
18400	15566.3858907593	51	2724	1284.284836975984	0.6402783975524141	9.286578251558351	46.49300187726276	0.2913010153642654 5	15452.362530353965	15680.314647797893	65529.061516719106
18500	16330.685215325 73	49	2637	1268.562617401902	0.6409861835130283	8.394041035546223	43.56061802586898	0.2656749780465503	16216.094835234448	16445.275595417 01	62186.899040385 42
18600	15518.666255658565	51	2659	1291.243078231905 7	0.6392896970142892	9.464579831284508	53.92976146477863 6	0.3023586818640033 3	15403.97447235 15	15633.358038696 63	65883.695237926 05	
18700	16098.820539106426	50	2630	1258.934131666311 3	0.6246163879091 39	8.660622712416059	46.59847395405909	0.2770043623022632	15984.467125955678	16213.173952257173	62981.312184016 85	
18800	15765.914785490993	50	2626	1287.381459102181 5	0.643476236516651	9.240170922922696	52.26979658546 5	0.2953425654593145 3	15651.198974756891	15880.630596225095	64397.445492404 1	
18900	15953.442639801673	50	2589	1268.413443476348	0.6361173203391385	9.057779655768 71	43.61405368572747	0.2928312187002 96	15839.272262194063	16067.613017409283	63444.860834088 46	
19000	16059.933262401419	50	2555	1272.634628086502	0.6283958694505666	9.014689005496987	46.22484398935986 5	0.2898012857367806	15945.365727270615	16174.500797532222	66263.387802650526	
19100	15455.377090362746	52	2658	1273.744393901556 4	0.6432248649816902	9.662261953503 43	45.18503161290306	0.3044517296162325	15341.296760450738	15569.457420274755	66263.864622220 15	
19200	16357.450293673957	49	2529	1271.442796948788 6	0.6349477083156784	8.766556896324953	46.10465550031 32	0.2845808690200644	16242.80024482219	16472.100342525726	62327.195302917 18	
19300	15395.532543264402	53	2662	1251.722820089214	0.6567199164331 51	9.759266550361 99	43.85774940880 3585	0.3111425463308694	15280.994908606883	15510.070177921922	66363.338504279 77	
19400	16197.741881738879	51	2569	1235.352142839963 3	0.6292437710171549	8.794482640308237	41.40631167289192	0.2845977759648991	16083.001330076038	16312.482433401 72	63033.183147565 46	
19500	15334.977019866203	52	2621	1274.873717805563	0.6504970098504352	9.855113238040 51	48.41271599743216	0.3216662837346 32	15220.340723595476	15449.613316136929	66323.444869606 96	
19600	15968.938092463302	50	2551	1261.990851754826	0.6472219414483311	9.224002524125174	46.20645557579289	0.3014104512827115	15854.179892363149	16083.696292563456	63131.414071249535	
19700	15729.013375199507	52	2598	1232.892995187692 3	0.6526697397291499	9.366115629209059	43.93379399082831	0.3034518554186224 6	15614.372103260 51	15843.654647138505	64141.663176297 26	
19800	15554.844513601 47	52	2583	1252.794104564234	0.6529411470175084	9.732862528013218	46.98947657230538 5	0.3143014851569 22	15440.920356558 59	15668.768671544 35	65174.080416711105	
19900	15921.687833963 92	52	2575	1229.111549223785	0.6323725540137617	9.183883682190714	42.81208957310626	0.2960704116786 2747	15806.947535031 93	16036.428132895911	63944.862802317 21	
20000	15491.626811965683	50	2458	1327.721465685079	0.6197613378896021	10.33681238271081	51.84008938985934	0.3386380795437681	15376.903468928222	15606.350155003145	66410.668000373 32	
20100	16584.927974339855	48	2340	1274.994407013 57	0.6274607889388593	9.007004875979238	45.49173229253234	0.3063760663633637	16470.211751979936	16699.644196699774	61231.939291571114	
20200	15415.207500808792	51	2481	1299.773207036801 5	0.6384906660649993	10.27535039928864 4	49.49847628172571	0.3370568843826066 7	15301.002147655012	15529.412853967433	66325.240821489 53	
20300	16054.131536013478	50	2440	1261.796070349581 4	0.6388148471979 49	9.428122073653336	43.82076845727342 4	0.3141980358854641 3	15939.460606902878	16168.802465124078	63122.647991440455	

20400	15765.355323371964	50	2423	1295.8679656085008	0.6311582937379536	9.9654291728461l	50.480488203509l3	0.3309253867564871	15650.790719720704	15879.879927023225	64830.13946022879
20500	15089.96838853921l	54	2550	1263.8964224624067	0.6093479432558274	9.9950420755337Sl	44.05136720610838	0.3216345390208237	14975.976970554215	15203.959806524212	68276.48152498533
20600	15381.241891485579	53	2532	1250.2443894501732	0.6012957247261637	9.6847962822033997	42.211316351137Sl	0.3136763193579148	15266.506711746708	15495.97707122445	66289.05167891046
20700	14779.879941074023	55	2612	1260.7936295397576	0.613622208973167	10.369084155353452	45.036670168180l3	0.3328356913590389S	14665.223011529553	14894.536870618493	69371.62130019732
20800	15429.687953265377	54	2553	1235.2384658867577	0.6094643033919485	9.45026549112835l	39.286824310780524	0.3044318821360279S	15315.132473592S9	15544.243432938163	66732.82605094394
20900	14512.422129139806	57	2640	1241.9042531825205	0.6126674956453497	10.678555203823503	44.589703844057l5	0.3423759226634762	14397.933026716048	14626.911231563565	70821.17375839956
21000	16250.980398726284	49	2263	1284.437318635523	0.5991060197701771	9.743055530189077	48.649264394038l5	0.3538850413976556	16136.20630839885	16365.754489053717	62966.303244493625
21100	15499.79134688896	52	2411	1263.4277103169882	0.6164206037160438	10.518117876793571	42.2116406271364l	0.3488890400842032	15385.093982696753	15614.488711081167	65732.07094221642
21200	15643.388828393376	52	2411	1253.1198303910971	0.628347082809l6	10.206783193877405	41.145516076025l5	0.3386131907331225S	15528.639204493507	15758.138452293244	65189.939370566906
21300	15773.778475261037	53	2401	1223.5792663844996	0.6102380833840394	10.097170858925825	40.163590806819Sl	0.3391483584370173S	15659.317104861399	15888.239845660675	64879.39410331449
21400	15299.13329678761l	54	2480	1222.5132483798395	0.633233433587l3	10.541508635399l9	38.72010221402201	0.3490376956703848l	15184.660689239961	15413.605904335272	66048.15970648231
21500	15908.691606928l7	52	2362	1232.306903208752	0.5999262602730998	9.8913207061923Sl	40.969762903066035	0.3320695050891426	15794.380839134317	16023.002374722624	64112.40815656498
21600	15304.058644108243	53	2415	1256.2949587405653	0.6241280511265007	10.933952453522l2	44.302966318912l7l	0.3638771727244262	15189.538922650163	15418.578365566324	66617.25138259293
21700	15949.675861462814	52	2350	1235.7235753644422	0.6091229067102192	9.93232536321466	41.243605742093166	0.3327679458086S154	15835.153487212947	16064.198235712682	64286.60143625673
21800	15166.39652686653	54	2392	1245.4601459156747	0.6330642465611l79	11.169354383457392	46.810325818904ll	0.3722656255795187	15052.334363198699	15280.458690534362	67289.039843875
21900	15598.526167743214	54	2396	1210.6251382677631297	0.6251386377631297	10.370067156240257	40.336598075904S3	0.3451205251427968	15483.940900189l5	15713.111434674529	65383.032987574
22000	15338.00342098571	54	2421	1220.4281725530512	0.6185266755230512	10.604979791550031	40.181672311919342	0.3540890269277787	15223.560054296906	15452.446787674513	65938.73134532825
22100	15601.571140315476	52	2311	1254.0972964454716	0.622081142288184l	10.814753704409386	42.904003150401l3	0.3667320813734652	15486.83729873255	15716.304981898404	65241.66338961922
22200	16079.485515721015	50	2237	1260.058172324143	0.5972385548936334	10.272485802843319	44.398151285370l2	0.3566749259809768	15964.682111947372	16194.288919494658	63031.94825875953S
22300	15467.713461385272	52	2279	1256.892677153372	0.6572752011066099	11.222342490278542	47.371724903134l3	0.3863509172789688	15353.118430573304	15582.30849219724	65392.26228267825
22400	16360.738166470524	49	2153	1261.9697871132l5	0.5824320248485928	10.200317678211198	49.798220217870l5	0.3564697127678107	16245.920946426177	16475.555386514872	61862.50885386113
22500	15165.729881728463	54	2334	1248.7624258249434	0.6617537817846245	11.483734365215l47	45.481792364746575	0.3854764921580055	15051.116300043155	15280.343463413l7	67468.1748073436
22600	15579.934130038968	54	2338	1217.5053426382223	0.6062202956472729	10.516979406664309	37.087787518407914	0.3540405762404967	15465.427196492677	15694.441063585258	65774.11717208415
22700	15315.914641842199	54	2356	1227.036942930945	0.6520766157617718	11.096626692501689	39.484165733202l4	0.3742035901424252	15201.142918202295	15430.686365482103	66291.91514119247
22800	15517.17351573715	54	2302	1211.3078716789425	0.6479489807709933	10.936223065351925	40.230011177903l5	0.3734473783343353	15403.229498423987	15631.117533050312	65447.29161278065
22900	15505.21149312361	56	2366	1186.9026901779523	0.6301997307177937	10.891764848196546	39.447354937627948	0.3634548899478826S	15390.662805571592	15619.760180675628	66488.44232474267
23000	15252.290555439866	54	2323	1223.3329409098185	0.6791081394887809	11.362092739327135	45.188312313613055	0.3892009111508521S	15137.70810989726	15366.873000982472	66083.03224661351
23100	16450.270912530465	48	2027	1296.0016784590932	0.6100106364195755	10.584288170639772	53.154272331240S5	0.3812876831607654l	16336.370386104936	16564.171443895993	62241.93709002l864
23200	14794.17036215403	54	2289	1277.1126305115579	0.670072217468273l	12.036542762280297	47.431111716400494	0.4096277349003935	14679.542472359444	14908.798251948614	68999.30967948376
23300	16198.964531387524	51	2133	1232.5251446328093	0.6427044092381651	10.668987695402308	43.040533673787l1	0.3759785712379146l	16084.303004887252	16313.626057887797	62891.48283758199
23400	14771.466659431992	56	2327	1229.3296864583892	0.676084662847359	11.781513575399748	41.582603168619S9	0.3988703980829983S	14656.773489200192	14886.159829663791	68873.81554983734
23500	15539.434317695208	54	2242	1213.3333330247l08	0.6598526371677162	11.135400917779197	38.042543195344365	0.3834974166287027l	15424.728883239637	15654.139752150779	65545.25980781025
23600	15379.796838312497	54	2256	1218.6530740027197	0.6930192342691535	11.302951696447336	38.174471763987l9	0.3906008518287433	15265.932689936597	15493.660986688397	65836.37714655219
23700	14863.953334107493	56	2315	1233.0059651342406	0.6627397192369677	11.834528536659l82	43.106910852490515	0.3996720506989248	14749.223557450721	14978.683110764265	69078.83388052638
23800	15853.455840337709	52	2178	1225.2395080980293	0.6529056478475107	10.887681760890699	41.675532443749	0.3840264178586123S	15738.945817995118	15967.96586268O3	63742.68833728868
23900	14540.738485534152	58	2368	1199.827774518692	0.6712460663319436	12.118705328959O4	40.548795432803786	0.4094883523278586	14426.081123467457	14655.395847600847	69623.49491960429
24000	15842.371167682002	53	2169	1205.7595177942715	0.6423312030224451	10.899894676125975	40.553686540310586	0.3878112293747502	15727.615327122343	15957.127008241661	63933.1742457069
24100	14648.82305706469	57	2314	1222.3349361250441	0.6656199918962351	12.224375976957399	45.40815126958203	0.4165351896722l054	14534.297770428957	14763.348343700423	69703.56220668231
24200	15185.720798312O3	56	2261	1205.0249965239948	0.6579148573316488	11.444215272988775	30.654722450477l2	0.3916872152429673S	15071.600813547922	15299.840783076137	67516.72874046718
24300	14893.342849237286	56	2295	1208.7685053701898	0.6592800959276335	11.829403276850659	31.514562087451875	0.4072446772121781l	14778.716088028475	15007.969610446096	67726.09103681137
24400	14879.63377405318	58	2284	1185.8167380728198	0.6717040333023946	11.961109532418629	34.936153310445206	0.4067950448135957	14764.957499439128	14994.309988867233	68813.1816336462
24500	15014.53498976871	58	2303	1165.44580605701	0.6477769071255726	11.519915960394686	29.415934855546833	0.3935923830293943l	14899.846273076693	15129.223706460727	67624.92169250973
24600	14573.916221658972	58	2300	1201.5355702523448	0.6687298902514329	12.423373543700862	34.444820485771l82	0.4240726530946252	14459.28704147253	14688.545401845415	69727.13090468508
24700	15517.534655619538	54	2129	1230.8039665168474	0.6643214453652996	11.304498697442268	46.408464858227326	0.3986594651579984	15402.871452341573	15632.197858897503	66492.64358834196
24800	14631.985346720623	56	2238	1232.4778437828552	0.6893459559702333	12.592847896786587	42.572407755232332	0.4393157726244466S	14517.292121996617	14746.678571444629	69052.91433290225
24900	15083.388848707769	56	2223	1201.6895540987812	0.6743711591657292	11.502370634399881	37.749767819370604	0.3996805813344326	14968.159758747322	15198.617938668056	67329.4122430137
25000	14793.28174432228	57	2241	1211.2350867778228	0.6652446459166711	12.307678666493932	38.312917759026775	0.4291751802089971	14678.641912275612	14907.921576368946	69073.45874986668
25100	14808.490671079044	58	2235	1185.8543380847273	0.6885559212408643	12.090534226094086	38.145837307474S3	0.4173630904305945	14693.921510412069	14923.059831746019	68816.89893539011
25200	14882.930620220064	58	2248	1174.670192227679	0.6699779278626l6	12.021452651380535	31.116182265937937	0.4163576544200918	14768.374775245322	14997.486465194806	68161.21709775478
25300	14597.846780313833	60	2303	1180.4744073982722	0.6802084917736699	12.651726249372066	28.955421985614116	0.4287698287222054	14483.179323179498	14712.514237448162	70852.43488080635
25400	15380.948914844923	54	2091	1221.8216649556468	0.6563149017562l37	11.729905447287l36	39.523424960l994	0.4280063812602638	15267.496690776054	15494.401138920792	66010.66369626953

25500	14420.54791696945	59	2294	1194.2535535957852	0.6775670529767467	13.140448254122761	31.6176608472118	0.44713477248830324	14306.022164373602	14535.073669565298	70496.0139975868		
25600	15492.37643384922	54	2091	1212.7899598502897	0.6493142477100717	11.594139960391322	38.33264912798738	0.41590540650002603	15378.411619398852	15606.341248299588	65526.54889392355		
25700	14499.65750205411	60	2277	1158.7728992509287	0.6818241540354991	12.843864632884939	29.7724247075472	0.44398207828197399	14384.893298383915	14614.421705724304	69557.27666357794		
25800	14829.196286068684	60	2259	1142.9128391208026	0.6692035071013198	12.140480352483602	26.706127566202763	0.41784654110638777	14714.474124681188	14943.918447456179	68601.56602747853		
25900	14647.206289060869	60	2267	1152.8283726979005	0.6860511336654368	12.545042115471812	27.996985022394377	0.43216125002160966	14532.524156932734	14761.888421189004	69191.24510292785		
26000	14579.741379390513	60	2272	1154.4143670448686	0.6850068333941686	12.693468123894872	28.459021470456996	0.43809173452301480	14465.08450678463	14694.398251996396	69292.46470188256		
26100	14819.34340460287	60	2248	1146.7036934474604	0.6692244855381078	12.289512739480712	27.790535209494277	0.42191037274041560	14704.628066897296	14934.058742308445	68833.42232914244		
26200	14353.114046458186	61	2293	1166.9358901896817	0.6903107733402449	13.193791344938424	30.100214114772772	0.45247909773896020	14238.852730871846	14467.375362044526	71209.23108200896		
26300	15494.6621261331	54	2020	1223.6115679200814	0.6722704889719106	12.03320684778	37.097643023639140	0.43527340450510354	15379.918256196399	15609.405996069801	66105.95028198496		
26400	14446.998417873681	61	2221	1158.2935341863733	0.6789492483839061	13.383324954171464	28.888215179836983	0.46878009364058437	14332.363806665533	14561.633029081830	70692.1720584902		
26500	15089.621925943055	58	2136	1156.3345214634528	0.6800293466680742	12.274878752253056	27.280156834672970	0.43668102804198236	14974.942590956573	15204.301260929537	67094.01534251506		
26600	14399.411256409223	62	2265	1140.8777137154834	0.6842647838540340	13.18098355931685	22.813000585102893	0.45172417820067730	14284.72571571065	14514.096797107795	70758.41669544291		
26700	14387.772070494595	62	2270	1139.701858427766	0.6449011441545835	12.929871391734302	21.396079393363408	0.44480450685001230	14273.051133570756	14502.493007418434	70697.46504619754		
26800	14737.920775723269	61	2223	1137.2363756282725	0.6501469954564921	12.633356953156248	23.757633847412002	0.44198661009016940	14623.285916686313	14852.555634760225	69400.62375339982		
26900	14345.190434369608	61	2253	1161.5317609891406	0.6661248602584320	12.54115641543652	26.598115880023958	0.46435764457759950	14230.599280357155	14459.781588382060	70884.91672610794		
27000	14782.304594368712	60	2189	1141.0440335682097	0.6489067369528376	12.54367148005188	26.582283068398510	0.43775018375973035	14667.579877845326	14897.029310892098	69089.42868813663		
27100	14064.432157713005	64	2274	1131.3189836021247	0.6714278571487402	13.830481534681681	26.577177892777130	0.47277323624386386	13949.657016631214	14179.207298794796	72430.4422180817		
27200	14972.389459000213	61	2179	1116.1471072854254	0.6485904840962705	12.410941203527413	24.119557124695902	0.43538131190778850	14857.70570264485	15087.073215355576	68117.10678030505		
27300	14317.222280653106	62	2213	1143.0865807533273	0.6805433146429103	13.704650823804199	20.857754942934346	0.47712775586696650	14202.580691979596	14431.863869326617	70902.14692716388		
27400	14817.617849868271	60	2126	1152.4149751346595	0.6543046788541624	13.084486286476139	23.347363465727987	0.46108914531523	14703.27700331153	14931.958696425012	69187.4129073116		
27500	14638.087721204434	61	2190	1137.9110032710819	0.6736614889743038	13.311957787365493	22.489435599134620	0.46349298974731146	14523.252164621068	14752.923277878	69446.11853701135		
27600	14372.025985321237	62	2197	1146.695261086075	0.6636021590185871	13.708959871556063	21.850909620215504	0.47531484132034	14257.363556141236	14486.688414501240	71131.18112967309		
27700	15009.085681950075	58	2086	1158.8666591287547	0.6454464226898179	12.791034911822893	24.55167109580762	0.46159530565397444	14894.368845728131	15123.802518172019	67252.33464881872		
27800	14129.864093406974	64	2235	1116.0240812052523	0.6774975807820557	14.054690020708202	19.556279302266947	0.49002654894968273	14015.12168416034	14244.606502653607	71464.17033524878		
27900	15194.652337390653	58	2031	1143.2602610676852	0.6491863545843535	12.579255048462096	27.512590970939947	0.46202374706512720	15080.103382151661	15309.201292629645	66348.98076035945		
28000	13932.1160387501	66	2277	1101.9834968145126	0.6674527862790359	14.357725265046323	22.436709364335872	0.50053221907398100	13817.361457278723	14046.870620221478	73564.78739233907		
28100	14696.456685078601	63	2185	1152.6327489430854	0.6532378030680641	13.069430281338484	20.565487206000727	0.45877357053323303	14581.734089362086	14811.179280795117	69435.36594781611		
28200	14264.256633580633	64	2215	1114.5365981626808	0.6537967061603075	13.802932203886678	19.999321350907096	0.47904182136587830	14149.507364448889	14379.005902712377	71369.8985030798		
28300	14375.341302746065	63	2206	1117.8381103907263	0.6624231349439792	13.787011331290481	20.457973863518525	0.47757926140400520	14260.99815100711	14489.68445448502	70461.19840941817		
28400	15006.335847927556	58	2023	1161.5396589982225	0.6415309335670155	13.323768020063778	25.689426455493724	0.48771803389000980	14891.567037179597	15121.104658675515	67399.02240280519		
28500	14009.817806455141	64	2211	1136.0125528044753	0.6640023987068325	14.39436803352398	21.023738293120810	0.49912171692802710	13895.12383239337	14124.511780516912	72737.24833008906		
28600	15099.674263116054	60	2015	1126.9001686743375	0.6526829055799009	13.078972092208030	23.806035360551892	0.47526158010302350	14984.964644744588	15214.383881487520	67644.74502293211		
28700	13873.976324850566	66	2234	1103.8128606676328	0.6793408255118648	14.693056093409446	18.05152112855506	0.51092365556246650	13759.843956903058	13988.108692798074	72883.62338341954		
28800	14878.784110182021	61	2075	1116.7712590902256	0.6755492811304188	13.334133055714942	20.929804380266013	0.47666260093966603	14764.299113438015	14993.269106926027	68154.73266325796		
28900	14034.074177452694	66	2224	1095.948114034928	0.6951096905028258	14.32940675013521	16.94592645115096	0.49726675344287	13919.370917074744	14148.777437830644	72362.75606300998		
29000	14321.123968192345	65	2207	1089.0445854374948	0.6822629051556184	13.872111857855414	14.946544656849918	0.48140421984159976	14206.322300241278	14435.925636143413	70819.34456062343		
29100	14392.355776070657	64	2175	1094.1218437464584	0.6801341353340139	13.784066291023851	17.403723497913958	0.48806072477475	14277.547421577587	14507.164130563728	70055.12839648552		
29200	13922.336404811847	67	2274	1088.1078837735618	0.6935360368728570	14.52651139769757	15.067022935811346	0.49661123111015	13807.61375140518	14037.059058218514	72931.53674037385		
29300	14604.416455969578	62	2100	1116.5244655822666	0.6784405943207551	13.477420552808601	22.81009733487093	0.48892465182071570	14489.802715256961	14719.030196682195	69259.26195536238		
29400	14425.115240120329	60	2024	1171.5428160775873	0.6945349381751004	14.847738766350128	28.616815397305263	0.54093327433752660	14310.633589302854	14539.596890937804	70326.44804610421		
29500	15332.210639037326	58	1896	1142.982151910094	0.7031849220139044	13.355006848826008	28.26490239343469200	0.50183681080363100	15217.391879497347	15447.029398577306	66327.88458282225		
29600	14214.109928240163	64	2089	1118.1742713096967	0.7008116518496995	14.843875650710487	21.00728095614191	0.53040047749891510	14099.369180334634	14328.850676145692	71599.4861307397		
29700	14860.512672533036	60	1960	1139.5994109314172	0.6969747844410620	13.959629816851955	27.882414114385840	0.51551397579211	14745.724146057084	14975.301199008987	68413.34801310596		
29800	14351.300215256158	64	2105	1100.2438016691192	0.7023111740888833	14.388380549395270	17.199983084933038	0.51622674693256230	14236.61086224461	14465.989568267707	70453.1021638846		
29900	14216.035025138543	65	2139	1096.7407612568354	0.7100957089530584	14.589191589850433	16.512382063274696	0.51482790307365880	14101.324484202692	14330.745566074393	71312.89680050398		
30000	15103.012220015244	58	1907	1154.9256041172061	0.6888218178992933	13.912802060799736	31.612691014924486	0.52370797465983540	14988.149019162229	15217.875420868259	67011.37058290225		
30100	14710.150868302085	58	1902	1186.5529376364429	0.6844891417482870	13.139369270091784	36.074166906358750	0.56940252010264960	14595.419470433253	14824.882266172446	68853.11616647113		
30200	15287.639271389182	57	1877	1158.4561930439731	0.6938545159598306	13.688166074048045	30.033198022338887	0.51365932938768480	15172.984321151562	15402.294221626802	66072.13942023092		
30300	14631.732519094841	58	1891	1198.874621796033	0.7048299974267052	15.42228810356274 2	39.22681239960403	0.57822686431116570	14517.048566855234	14746.416471334449	69577.85450609 3		
30400	14900.835172776067	61	1952	1109.5399841134156	0.7069593000547184	14.130813673543955	26.589242048587835	0.51963107279439980	14786.006771833909	15015.663573718226	67717.55400945283		
30500	15396.092341728418	55	1745	1204.3247333470215	0.6467095589834809	14.861372557551473	41.006140559064910	0.58174776023080480	15281.352239919135	15510.8324435377	66271.11875266653		

30600	15381.78869033355	55	1749	1206.1986187039718	0.6603373083873083	14.671341004942493	41.22123527296394	0.5724853110706389	15267.047447383347	15496.529933283751	66374.9306986960
30700	15461.072221444376	55	1754	1193.8017814807451	0.6571175195965104	14.526894279625493	38.904425235657456	0.5642555163175972	15346.299172426006	15575.845270462745	65694.1912098021
30800	15543.522172405274	52	1656	1252.414096402038	0.6524737222639986	15.098683283187862	52.870934518449445	0.6096033200530224	15428.700355230994	15658.343989579553	65167.2508082902
30900	15976.405263703957	52	1633	1241.3859150909386	0.5916365554424641	14.136013607137118	53.24201722980708	0.5670499779398961	15861.545171130367	16091.265356277547	64583.0383112900
31000	15045.828710621388	54	1697	1269.179724513458	0.6941391441825754	15.89517772860797	57.12221537473471	0.6258807204137604	14931.114050246399	15160.543370996378	68574.2770252253
31100	15809.737411994112	52	1651	1234.4595072942263	0.5936468385911572	14.070487106223698	49.20325008597595	0.5675569056393073	15695.150542137071	15924.324281851154	64224.4756346328
31200	15359.035694035183	52	1646	1273.8754183494825	0.6779809111600098	15.465692095246995	57.76467462594258	0.6225917589239998	15244.289553524035	15473.781834546331	66278.1093774998
31300	15214.038885331873	58	1772	1157.7058599231032	0.6234713950507934	14.792581873624355	37.76724261135050	0.5720052433718089	15099.431149599022	15328.646621064723	67183.2865363847
31400	15322.9112672903	56	1729	1179.3152712748179	0.5978324945503185	14.866587496285932	44.02370375140835	0.5871939373895022	15208.044698359381	15437.777836221218	66078.9729107780
31500	15052.877022386252	56	1727	1201.7224399698682	0.6392522084469242	15.360513711223987	43.38046764920418	0.6072023096897959	14938.017679356293	15167.73636541621	67327.579778078
31600	15739.514122591378	54	1646	1203.7271671807134	0.5441248270201339	14.576132949499062	43.76389799570896	0.5831625467757047	15624.726167034058	15854.302078148698	65032.6390202522
31700	14282.105343863353	60	1858	1177.2400336148742	0.6059285410221971	15.897440421763806	35.51219145549016	0.6075342849018927	14167.351321871576	14396.85936585513	70670.6296297530
31800	15819.089452596725	54	1646	1195.5615051579964	0.5402970632799217	14.30274688141194	40.92804001442065	0.5724571614399661	15704.257453163995	15933.921452029455	64591.0964215241
31900	14233.091225773283	60	1861	1180.3945981992076	0.6222908374926808	15.995860192310452	37.86097656716165	0.6117671663220637	14118.263788665943	14347.918662880624	70850.6425540637
32000	15373.423343412042	56	1728	1175.1786272410707	0.5869210022704325	14.762669198537806	39.54373754520426	0.5830906284091371	15258.65262082668	15488.194065997403	65839.1749423364

Appendix IV

Sample of debug log

```
18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: Program started

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: BufferManager.BufferManager
(1000000,10000,DBMSLogger)

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:50 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated
```

```
18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 10240000B of memory allocated

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: 100 segments created. Freesegment: 100

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: Checkpoint.checkpoint
(buffermanager, timerobj,properties, loggerobj)

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO:
TrabsactionController(1000000,10000,5,1000000,bu
fferManager,dbControll,loggerObj)

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: TransactionControll.run()

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: Checkpoint.run()

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(0)

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: Segment0 is locked

18.mai.2005 15:01:51 DBMSLogger logMessage
INFO: RDMA will take 10.0367431640625
milliseconds

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: createThreads(5,1000000,bufferManager)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment0 is unlocked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: All threads have been started

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(1)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0.run()

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1.run()
```

```
18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2.run()

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3.run()

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4.run()

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment1 is locked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: RDMA will take
10.0367431640625 milliseconds

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(true)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: CopyOnWrite is activated for the0th time

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Copy segment: 1 to segment 100

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment1 is unlocked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Free 100-Locked 1

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(2)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(false)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment2 is locked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: RDMA will take 10.0367431640625
milliseconds

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(true)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: CopyOnWrite is activated for the1th time

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Copy segment: 2 to segment 1

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Free 1-Locked 2

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(false)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment2 is unlocked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(3)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment3 is locked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: RDMA will take
10.0367431640625 milliseconds

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(true)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: CopyOnWrite is activated for the2th time

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Copy segment: 3 to segment 2

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Free 2-Locked 3

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment3 is unlocked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(false)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(4)
```

```
18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment4 is locked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: RDMA will take
10.0367431640625 milliseconds

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(true)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: CopyOnWrite is activated for the3th time

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Copy segment: 4 to segment 3

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Free 3-Locked 4

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(false)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment4 is unlocked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(5)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment5 is locked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: RDMA will take
10.0367431640625 milliseconds

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(true)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: CopyOnWrite is activated for the4th time

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Copy segment: 5 to segment 4

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment5 is unlocked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Free 4-Locked 5

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(6)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread2 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(false)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment6 is locked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: RDMA will take
10.0367431640625 milliseconds

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(true)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: CopyOnWrite is activated for the5th time

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Copy segment: 6 to segment 5

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Free 5-Locked 6

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread4 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread1 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: setBusy(false)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread0 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Thread3 is sleeping

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment6 is unlocked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Checkpoint.WriteSegment(7)

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: Segment7 is locked

18.mai.2005 15:01:52 DBMSLogger logMessage
INFO: RDMA will take
10.0367431640625 milliseconds
```