The ClustRa Telecom Database: High Availability, High Throughput, and Real-Time Response

Svein-Olaf Hvasshovd
Telenor Research
N-7005 Trondheim, Norway
Svein-Olaf.Hvasshovd@tf.telenor.no

Øystein Torbjørnsen
Telenor Research
N-7005 Trondheim, Norway
Oystein.Torbjornsen@tf.telenor.no

Svein Erik Bratsberg
Telenor Research
N-7005 Trondheim, Norway
Svein.Bratsberg@tf.telenor.no

Per Holager
SINTEF DELAB
N-7034 Trondheim, Norway
Per.Holager@delab.sintef.no

Abstract

New telecommunication services and mobility networks have introduced databases in telecommunication networks. Compared with traditional use of databases, telecom databases must fulfill very tough requirements on response time, throughput, and availability. ClustRa is a telecom database prototype developed to run on standard workstations interconnected by an ATM switch. To meet the throughput and real-time response requirements, ClustRa is a main memory database with neighbor main memory logging. Transactions are executed in parallel. To meet the availability requirements, we use a 2-safe replication scheme over two sites with independent failure modes, a novel declustering strategy, early detection of failures with fast takeover, and by on-line self-repair and maintenance. This paper gives an overview of ClustRa and includes a set of performance measurements.

1 Introduction

Digital switches are monolithic units supporting plain old telephony service (POTS). They have over time been stuffed with increasing amounts of functionality, e.g., for management of routing, terminals, subscribers and charging. A digital switch typically contains millions lines of software. It is not allowed to be out of service for more than two to three minutes per year, corresponding to availability class five [GR92]. The large amount of software combined with the required high availability results in long lead time for introduction of new services and a dominance of switch suppliers over service providers.

Intelligent networks (IN) were introduced in the 80's to support new types of telecom services, e.g., terminal mobility (UPT), virtual private nets, and credit card calling. Most IN services are required to have the same service availability as POTS. IN are designed for rapid development and deployment of new services and to give service operators control over service development. To obtain this goal, services are built by service programs using basic functions supported by specialized servers (SCPs) in the network. The effect is that functionality that was buried invisibly inside digital switches becomes open, modularized, and allocated as servers on multiple platforms. One entity that appears in this architecture is the telecom database.

The classical use of telecom databases is in various types of call routing, where transactions read one record, demand 5 to 50 milliseconds response time, availability class five, and 10 to 10.000 TPS. Examples are mapping from phone number to terminal in a digital switch, and from a universal phone number to a physical phone number in UPT. Update transactions are less than 10% of the transaction volume, and demand from 50 milliseconds to a few seconds response time, with availability class five. Mobile telephony implies update transactions with 10 to 20 millisecond response time because a user should experience no longer glitches than 100 milliseconds when a mobile terminal is handed over from one switch (MSC) to another. Durable connections which imply crash atomic call sta-

Proceedings of the 21st VLDB Conference
Zurich, Switzerland, 1995

tus, demand update transactions with similar response time as mobile telephony.

ClustRa is a telecom DBMS kernel designed to meet the total combination of requirements from IN and mobility networks. As a consequence, all performance requirements refer to TPC-B-like transactions, i.e. transactions with the same semantics as TPC-B [Gra91] transactions, but with a data volume adapted to the application. The following are the main project goals: 1) Response time for TPC-B-like or lighter transactions of maximum 15 milliseconds for at least 95% of the transactions; 2) Scalable throughput with an upper limit of at least 1000 TPC-B-like TPS; 3) Class five availability.

This unique set of requirements are approached by a combination of old and well known techniques, together with new techniques developed primarily to meet the availability goal. To meet the response time, ClustRa employs a main memory database for real-time data, main memory logging, and parallel intra-transaction execution. A parallel database design is used to achieve scalable transaction throughput. To meet class five availability we use a 2-safe replication synchronization over two sites, and on-line self-repair. ClustRa uses the relational model, but can also support object oriented models. The system is also designed to support traditional on-line and decision support transaction types through the use of traditional database buffering and a flexible record structure.

The organization of the paper is as follows: The state of the art is briefly surveyed in Section 2. Section 3 presents the ClustRa architecture and how it executes distributed transactions. The ClustRa availability management is presented in Section 4. The detection and masking of node failures are emphasized. Section 5 gives an overview of the ClustRa log and recovery method. Measurements of transaction response and throughput together with take-over time are presented in Section 6. Section 7 concludes the paper.

2 State of the Art

Telecom databases have mainly been developed by switch manufacturers. These systems have barely been documented in the research literature. They were tailored to the needs of routing applications in digital switches. As a consequence, they support very fast reading of a few records, and most provide dirty read. Some systems are pure main-memory databases, others have background disk support. Update transactions involve write-ahead logging to disk, and the response time is therefore longer than a disk access. The throughput rate for update transactions is rather low. Some systems are centralized, some are parallel. The parallel systems seem to give scalable throughput growth. They differ on availability attention and implementation. Some rely entirely on fault-tolerant hardware, others have implemented this in software using multiple loosely synchronized main memory replicas of tables. A hot stand-by replica becomes primary in case the primary fails. Automatic repair is realised by producing a new replica for one that has been lost. On-line software upgrade is not supported. The relational model is used by most systems.

New telecom databases have appeared over the last years [Ahn94]. One of these is the Dalí system from AT&T Bell Labs [JLRS94]. This is a single node site main-memory system with background disk support that is tailored to routing applications. It supports more flexible record structures than most older products. It uses a 1-safe hot spare system with a system as the atomic failure unit. By utilizing hot spares, Dalí indicates ability to support schema and software changes without bringing the entire system down. Dalí does not have throughput scalability given its current centralized architecture, nor does it support on-line automatic repair, which is imperative to achieve class five availability.

Smallbase is a telecom database developed by Hewlett-Packard Laboratories [HLNW94]. The system is designed for the throughput and response time characteristics of IN transactions. A main goal is to achieve transaction scale-up using commodity hardware and Unix. Like Dalí, Smallbase uses a 1-safe hot spare system.

TDMS is developed by Nokia for use in switching and mobile systems [Tik92, Tik93]. The system runs on dedicated hardware and basic software. The focus is on response time and throughput of read transactions. Throughput scale-up and high availability have not been catered for beyond the use of fault-tolerant hardware.

Commercial SQL databases are used within IN and some mobile applications. Some SQL systems meet throughput and response time requirements for read-only transactions, but their applicability to mobile and switch applications are limited by their longer response time for update transactions caused by disk logging. Availability has been achieved through system pairs and manual repair. Limited attention has been given to aspects like on-line schema upgrades.

3 Database Architecture

3.1 Platform

For fault-tolerance, the ClustRa database management system uses a shared-nothing hardware model. Each *node* of the database system is a standard Unix workstation with a 64.0 (Sun Sparcstation 5/85) or

50.2 (Sun Sparcstation 10/40) SPECint92 CPU and 256 MBytes of RAM. UNIX (SunOS 4.1.3) is the operating system at each node. Inter-node communication is via an ATM (FORE Systems ASX-200) switch with a capacity of 100 Mbits/sec per connection through the switch. The purpose of the project is partly to show that it is possible to meet the requirements of a telecom database using standard, off the shelf hardware and operating system.

Each node is an atomic failure unit. Nodes are grouped into *sites*, which are collections of nodes with correlated probability of failure. Sites are failure independent with respect to environment and operational maintenance. Logically, the database system consists of a collection of interconnected nodes that are functionally identical and act as peers, without any node being singled out for a particular task. This improves the robustness of the system and facilitates the maintenance and replacement of nodes. Figure 1 shows an architecture with two sites, each having four nodes. Each site has a replica of the database. Each node at a site is connected to an ATM switch, which again is connected to the switch at the other site. The switches have external connections.

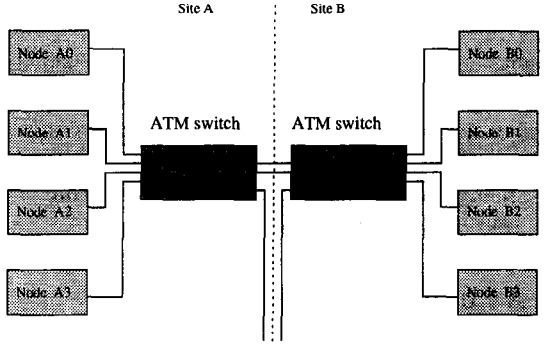

Figure 1: An architecture with two sites.

3.2 Traditional and Main Memory-Based Server

ClustRa is a traditional database server, in the sense that it manages a buffer of data with a disk-based layout in blocks; it has a B-Tree access method, a semantically rich two-phase record locking protocol, and it has a two-level logging approach. However, it is main memory-based in the sense that tables may be declared to reside in main memory. Unlike pure main memory databases [HLNW94, JLRS94], this allows for many classes of queries and transactions, not limited to those requiring real-time response. This is important in a telecom database, due to multiple services having diverse characteristics with respect to response time and data volume.

High throughput is achieved by a distributed architecture. A table may be distributed onto different nodes by horizontal fragmentation, according to either a hash or a range partitioning algorithm. High availability is achieved by replication over several sites. We are using an asymmetric replication scheme, where there always will be one primary replica of a (horizontal) fragment of a table, but there may be several hot stand-by replicas. The node of the primary replica will always be the one executing the request for a specific record found in that replica. The hot stand-by replicas will be kept consistent by redoing log records that are shipped from the primary node. Each node in the system may be primary for some fragments, and hot stand-by for other fragments. This facilitates load balancing both during normal processing and during node failure where *takeover* must take place, i.e. the hot stand-bys will become primary. The number of hot stand-by replicas are dependent on the desired availability level. For our requirements it is enough with one hot stand-by replica [Tor95].

ClustRa provides a basis for a relational system. It supports variable length records identified by primary keys and organized in tables. Records may be accessed by primary keys or sequentially, and they are stored in fixed sized blocks according to the block size in the underlying secondary storage. Tables are stored in files, which currently are organized as B-Trees. This holds also for resident, internal administration data, like the free block management, the resident part of the distributed log, and the file directory, which is a mapping from file identifiers to root block identifiers. By using B-Trees also for internal data structures, we have reduced the code volume of the system. We have chosen to use a B-Tree access method due to its generality – it gives sufficient performance both with respect to sequential and direct record access.

To access the database we use an internal code format, which express a rich set of record, algebra and cursor operations. In addition, this code format is used in messages sent between the processes. Thus, it is also capable of expressing log records and information about transaction processing.

The buffer manager holds copies of blocks residing on disk with three different priorities:

- *Real-time*: the block always resides in main memory.
- *Random*: the block will be accessed randomly and is kept in the buffer according to a LRU policy.
- *Sequential*: the block will be accessed sequentially. This is a hint to the buffer manager to

prefetch the next blocks in the same file and to make used blocks available for replacement.

3.3 Runtime Architecture and Transaction Execution

On each node there is a number of services: A transaction controller, a database kernel, a node supervisor, and an update channel. These services may be built (during make) either as separate UNIX processes, or as our own light weight processes (threads) inside one UNIX process. The transaction controller receives requests from clients to execute certain precompiled procedures, or it receives user code which it compiles into the internal code format. It coordinates transactions through a two-phase commit protocol. The kernel has the main database storage manager capabilities, like locking, logging, access methods, block and buffer management. It receives code to be interpreted from a transaction controller. The update channel is responsible for reading the log and for shipping log records to hot stand-by nodes. The node supervisor is responsible for collecting information about the availability of different services, and for informing about changes.

Figure 2 illustrates the 2-safe [GR92] execution of a simple transaction by showing the processes involved. There are two sites with two nodes each. The transaction controller (T0) receives the client request, and thus become the primary controller. It has a hot stand-by controller (T2) on another node, which is ready to take over as primary if T0 fails. T0 uses the distribution dictionary to find the nodes holding the primary and the hot stand-by replicas of the records in question. For this particular transaction, K1 has the role of a primary kernel and K3 the role of a hot stand-by kernel. T0 sends the operations to be executed *piggybacked* on the start transaction command to K1. Simultaneously, it sends K3 a start transaction command together with the number of log records to receive from K1. The update channel (U1) reads the log of K1 and ships the log records for this transactions to K3, where they are stored in the log and redone. When T0 has received ready (and possible return values) from both kernels and an ack from the hot stand-by controller, it gives an *early answer* to the client. The second phase of the commit includes sending commit messages to the two kernels involved. When both kernels have responded to T0 with done, the transaction is removed from T2 and T0.

The illustrated transaction is simple, because when a transaction accesses several records, there may be several primary and hot stand-by kernels executing the transaction in *parallel*.

Internally each process is organized as a set of

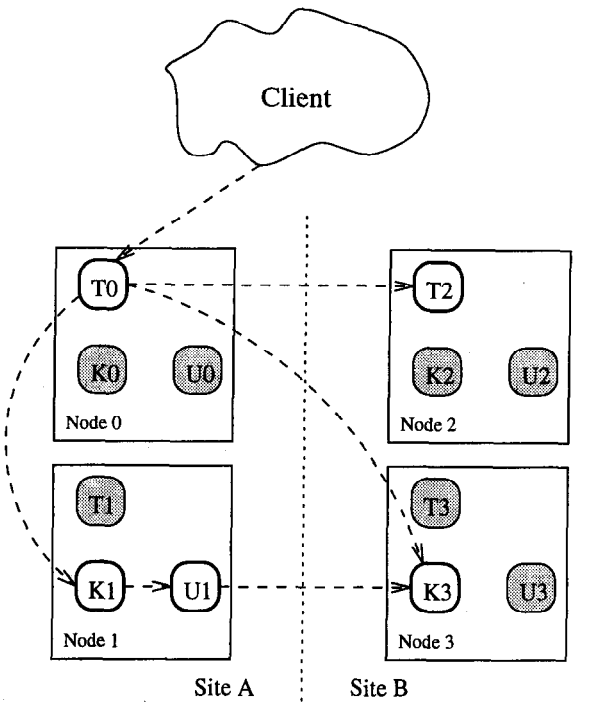

Figure 2: Processes involved in a simple transaction.

threads, which are handled by our own scheduler. Threads facilitate multiple users in the system without any significant overhead, as well as waiting conditions in the communication and for disk access. Threads are non-preemptive, which are used to ensure that access to shared data is synchronized.

4 Availability Management

High availability is based on data replication and allocation of primary and hot stand-by fragment replicas to nodes with independent failure modes. Additionally, ClustRa balances the load on the nodes both during normal operation and failures, and it supports *self-repair*.

To discover failed services or nodes, ClustRa applies an I-am-alive protocol internally inside a node and between nodes. All processes on a node are polled repeatedly by a node supervisor to detect their state. The I-am-alive protocol between the nodes is organized as a circle. Each node sends and receives I-am-alive messages from both its neighbors in the circle. In addition to discovery of failed nodes and services, the I-am-alive protocol is used whenever a node is changing the set of provided services. It is also used to determine which unavailable nodes each available node is responsible for attempting to restart at regular intervals.

The I-am-alive system is fully distributed with no

centralized knowledge. Hence, a *virtual node set protocol* is used to maintain a consistent set of available nodes [ESC88]. If consecutive I-am-alive messages are missing from one of the neighbors in the circle, the protocol is activated. The node supervisor sends a build node set message to all known nodes. Other nodes should respond with their services. If a node has not responded within a certain number of resends, the node is assumed to be down, and a new node set is distributed.

For high availability it is important to perform a fast take-over after a node failure. Given the low probability of lost messages and the short message queues in the ATM technology, we minimize the number of missing I-am-alive messages before the virtual node set protocol is activated. This shortens the interval from a failure happens to it is reacted upon. To minimize the delay caused by updating the distribution dictionary, it is cached at each node and the current node set is used as a filter to it. The distribution dictionary itself is modified outside critical path. To minimize unavailability of replicas changing their role to primary, locks are maintained (but not effective) to the hot stand-by replicas. These replicas become available immediately after redoing the hot stand-by log records arriving before the new node set. In-flight transactions being active at a crashed node are rolled back, while those losing their hot stand-by controller continue. To minimize the window where the system is vulnerable to double faults, the log where only one replica exists, is copied to a node at the other site. Together with the main memory logging to two sites with independent failure modes, this is as reliable as traditional logging to disk [Tor95].

After a takeover, a recovery of the failed node is started. If it does not recover within a certain period of time, ClustRa starts its *on-line, non-blocking self-repair*. New replicas of its fragments, possibly subfragmented, are produced fuzzily, and they are then caught up with the use of distributed logs [HST+91].

One of the main goals of ClustRa is to achieve high availability and load balancing, both during normal processing and upon failures. To fulfill this goal, a new strategy for placement of replicated fragments, *minimum intersecting sets declustering*, has been developed [Tor95]. The principle of this methodology is that the largest cardinality of any intersection of the sets of fragments allocated to different nodes, should be minimized. It achieves low unavailability by fast takeover and self-repair. A takeover is made fast by tuning the system to not involve more nodes than necessary. Self-repair is made fast by exploiting spare capacity of non-failed nodes.

5 Logging and Recovery

5.1 The Distributed Log

ClustRa combines two logging methods: a distribution transparent method for record operations and a physiological method for node-internal operations. The *neighbor write-ahead logging* developed in [Hva92] is applied to record operations. This is a main-memory logging technique where transaction commit does not force log to disk. A log record must be written to two nodes with independent failure modes, i.e. neighbor nodes, before the effect of an update operation is allowed to be reflected on disk, and before the transaction commits. The log shipping serves in addition as the loose replication synchronization scheme [Cri90].

The idea of this scheme is to save the delay in forcing log to disk at commit time. We do this by taking advantage of modern communication technology, where the delay in writing to main memory on a neighbor node is much lower than the delay in writing to disk. Since we exploit the log shipping for replication as well, this comes almost for free. The distributed logging uses a compensation log record (CLR) policy. CLRs are produced by the primaries and contain complete redo information and reference to the log record it compensates. Only a single CLR can be produced per non-CLR [MHL+92].

To allow subfragmentation of a hot stand-by replica as compared with its corresponding primary, both redo and undo use *logical*, i.e. primary key-based record access. This allows for encapsulation of block size and access method in each node. Of the same reason state-identifiers reflecting the sequence of operations executed to a record are connected to records instead of as traditionally to blocks. State-identifiers are generated at the primaries with unique values within a fragment. They are included in the log records and are reflected to the hot stand-bys through redo processing. Therefore, two replicas of a record with the same state-identifier reflect the same state independently of the fragment, replica, and node the record is located at.

Another strong point of the replication scheme, is that no concurrency control is needed on hot stand-by fragment replicas, because the execution order is the same to a hot stand-by record as to its primary. However, fragment shield locks are used on hot standby fragment replicas, because during take-over, when hot stand-by fragment replicas do not yet reflect all pending redo operations, new transactions should wait.

5.2 The Node-Internal Log

Node-internal operations for access methods, free block management, and file directory are implemented

transactionally. These operations are logged in a node-internal log. A physiological, single CLR policy with just reference to the compensated non-CLR is used [GR92]. The node-internal log is disk-based and is not shipped to any other node. To be able to undo a node-internal operation a node-internal log record must be produced in main memory before the corresponding operation is executed. The log record must have been flushed to disk before the effect of the operation is reflected on disk. A sequence may be imposed on writing blocks to disk to avoid logging records moved in a block split. Committing a node-internal transaction does not involve any block flush, because redoing the distributed log to the node will establish an equivalent node-internal state. This synchronization protocol between the distributed and the node-internal logging avoids altogether introducing log-related disk flushes in the time-critical transaction execution.

To support fast node crash recovery and efficient log garbage collection, ClustRa uses a fuzzy checkpointing algorithm in combination with a steal and no-force buffer management policy. Transactions are allowed to go on while the checkpoint is made, only a single block is read-latched at a time with copy-on-write option while they are inspected and possibly flushed to disk. The checkpointing limits the redo work, by allowing the redo recovery to start at the penultimate checkpoint in the log. However, the undo work must go back in the log until the active transaction table is empty.

5.3 Node Recovery

The recovery method of ClustRa distinguishes among different degrees of corruption at a node. When the disk is corrupted, the complete database is *reloaded* from other nodes. When the contents of main memory is garbled, a recovery based on log records and database from disk is done. When only parts of main memory are corrupted, a recovery based on main memory is done. The decision on using disk or main memory recovery is based on checksums on the internal administration data. If the checksums on the buffer access structure, the log access structure, and the lock structure are found to be in order, a main memory recovery may be done. A node recovery based on main memory involves just undoing the eventual ongoing node-internal transactions and undoing the record operations that were not reflected on any other node before the crash occurred. A disk-based recovery performs a redo followed by an undo recovery from the stable node-internal log, before a redo recovery is executed based on the distributed log shipped from nodes with primary fragments for those stored at the recovering node. If during a main memory-based recovery a block is found to be corrupted upon access, i.e. records or administration data inside a block is broken, a partial recovery is performed by reading the block from disk, redoing the node-internal log regarding this block, and then the distributed log.

6 Measurements

All measurements presented here were taken on Sun Sparcstation 5/85's, except the scaleup tests, where the first four nodes are singleprocessor 10/40's and the last four are 5/85's.

6.1 Response Time

We have performed some response time measurements for TPC-B-like transactions: three updates and one insert in different tables. Figure 3 shows the key numbers for both the single- and multi-process versions of the system. The statistics is gathered over a period of 80 seconds, running a total of 6395 transactions in the singleproc version and 5674 in the multiproc version. Figure 4 shows the response time distribution for the singleproc version. This test is run on a two node configuration, where one node acts as primary and the second acts as hot stand-by. Thus, the test involves distributed 2-safe two phase commit processing, where a hot stand-by replica is ready to take over in case the primary goes down. All response times are measured in milliseconds. From these figures we

Version	Avg	Min	Max	$\% < 15ms$
Singleproc	9.77	8.81	73.6	99.5
Multiproc	11.64	9.26	69.9	98.9

Figure 3: Response time measurements for TPC-B-like transactions.

see that the response time requirement that 95% of TPC-B-like transactions answer in less than 15 milliseconds, is met on this small configuration. In the singleproc version 73% of the transactions have a response time between 9 and 10 milliseconds. There is a small group of transactions taking around 60 milliseconds. We assume these transactions to be delayed by some regularly scheduled UNIX daemons.

The response times are approx. 1.87 milliseconds better in the singleproc version. This is due to less work done by the operating system. We avoid some process switches and overhead in node-internal interprocess communication.

The communication consumes a large portion of a transaction's response time. A remote message passing takes about 0.8 millisecond. There are four remote and

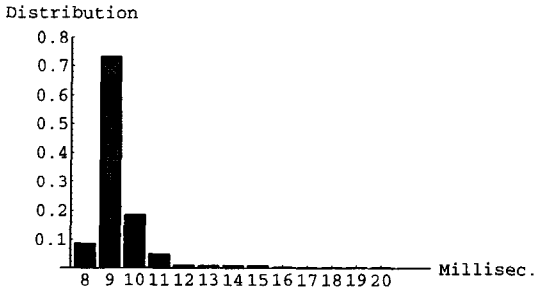

Figure 4: Distribution of response times for TPC-B-like transactions in the singleproc version.

two node-internal message sends included in the critical path of the transaction in the singleproc version. These remote sends alone consume 3.2 milliseconds of the response time. Outside the critical path, there are another two remote and two node-internal messages sent before a transaction responds. Including the commit processing are ten messages sent remotely, and six node-internally during a transaction execution.

The primary node has a CPU utilization of 70%, while the hot stand-by is 35% loaded. This load distribution is explained by the primary controller having a higher load than the kernels and the hot stand-by controller, because all internal messages are within the primary node, and because the client communication hits the primary.

6.2 Throughput

Figure 5 shows the measurements for throughput and average response time when we load the server with several parallel transactions. The window size is the number of parallel transactions. These measurements are taken over a period of 80 seconds. The increase in response time is an effect of queuing. From these figures we can see that this configuration has a maximum throughput of 104 2-safe TPC-B-like TPS in the singleproc version. With an acceptable response time, we can get 83 2-safe TPC-B-like TPS out of this configuration. We could have traded throughput with response time by coalescing many transactions into the same messages, and by using a grouped neighbor write-ahead protocol.

Version	Val.	Window			
		1	2	5	10
Singleproc	TPS	83.3	102.3	104.5	104.9
Multiproc	TPS	72.9	77.1	83.2	81.7
Singleproc	Resp	9.43	16.58	38.4	77.7
Multiproc	Resp	11.36	21.3	49.5	93.2

Figure 5: Throughput and response time as a function of window size for TPC-B-like transactions.

Nodes	TPS	Eff.	Response time		
			Avg	Min	Max
2	114	1.00	14.6	10.5	92
3	144	0.84	17.1	8.3	110
4	172	0.75	19.3	9.8	40
5	211	0.74	19.7	8.0	117
6	243	0.71	20.7	8.0	144
7	281	0.70	21.0	7.7	146
8	315	0.69	21.5	7.8	147

Figure 6: Scaling up in the two-updates transactions with high load.

To measure the scale-up of the system we implemented a transaction updating two records in two different tables. The tables were fragmented to all nodes, both as primary and as hot stand-by. To be able to run with an odd number of nodes without getting skewed load, we allocated fragments in a circle. Node 0's primary fragments have hot stand-bys on node 1. Node 1's primary fragments have hot stand-bys on node 2, and so on. To scale the load as well, each node has a separate client. Thus, when we add a node, we also add a client.

Nodes	TPS	Eff.	Response time		
			Avg	Min	Max
2	66	1.00	12.0	7.3	89
3	99	1.00	10.9	6.5	85
4	129	0.98	12.1	7.3	65
5	159	0.96	12.9	7.1	112
6	186	0.94	13.6	6.8	129
7	216	0.94	14.0	6.5	139
8	242	0.92	14.0	6.5	139

Figure 7: Scaling up in the two-updates transactions with medium load.

Figure 6 shows the throughput and response time as a function of the number of nodes when we run trans-

actions back-to-back, thus loading the server heavily. Figure 7 shows the same when we let the clients sleep some milliseconds between each request, causing less load on the system. In both tables the statistics for each row is gathered over a period of 40 seconds. The efficiency columns use two nodes as reference for throughput. Figure 8 illustrates the throughput graphically, where the dark plot is the heavy load and the lighter plot the lighter load. We can see that the throughput seems to scale linearly.

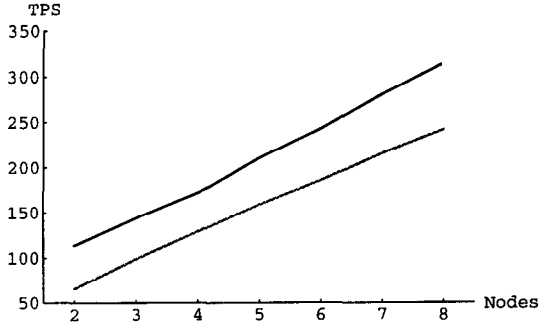

Figure 8: Close to linear scale-up.

The TPC-B-like transaction was not used for the scale measurements, because it touches too many records per transaction compared with the number of nodes in the current version of the laboratory. Due to the high cost of communication, there is a noticeable saving for the transaction when the two records are on the same node. When adding more nodes, we expect the response time and efficiency to stabilize, because the locality effect disappears. An indication that this is true, may be seen in the asymptotic behavior of the measured average response times and the fairly constant minimum and maximum response times.

6.3 Takeover

Fast takeover is necessary to ensure high availability. To measure the takeover time, we instrumented the client to register the interval the system was unavailable. In our measurements we used a two node configuration. One node holds the primary kernel and the hot stand-by controller. The other holds the hot stand-by kernel and the primary controller. The former node was stopped by sending it a UNIX signal.

The client sends transactions back-to-back to the controller and receives either committed or aborted status. The time is taken between the receipt of two committed transactions, where there was a node crash in between.

Figure 9 shows the takeover time measured both for the singleproc and the multiproc versions of the system. In these measurements the I-am-alive messages are sent between nodes every 50 millisecond. If a neighbor node in the I-am-alive circle has not sent any I-am-alive messages within 100 milliseconds, the virtual node set protocol is started. The node supervisor discovering the lacking I-am-alive message builds a new node set by asking all known nodes about their services. The nodes which have not responded within 50 milliseconds, is questioned once more. If they have not answered within another period of 50 milliseconds, they are assumed to be down, and a new node set is distributed. In average, the two phases take 225 milliseconds (125 + 100). The rest of the time is used in the takeover itself.

Version	Count	Avg	Min	Max
Singleproc	5	330	265	379
Multiproc	5	387	357	414

Figure 9: Takeover time (milliseconds) measured from the client.

7 Conclusions and Further Work

There are several other approaches (briefly surveyed in Section 2) addressing the response time and throughput requirements, but we do not know of any other approach which address all three requirements of response time, throughput, *and* high availability. The ClustRa project has during the first year approached two of its main goals for response time and throughput, and it has partially met the third goal for high availability. The *response time* requirement is met on small configurations of the system. The main technique used to meet this goal is the main memory database *with* main memory logging. Main memory logging is possible by writing the log to the main memory of another node with an independent failure mode. The current system is CPU-bound due to high communication costs. Thus, to meet the response time goal also on larger configurations, we are porting the system to workstations with faster CPUs, and we are optimizing the ATM drivers. The *throughput* seems to scale linearly. Thus, we assume the throughput goal to be met by adding more nodes. Fast take-over is achieved, and *high availability* will be achieved by having two

sites, minimum intersecting sets declustering, on-line self repair, and system maintenance.

Currently we are addressing takeback, where a failed node recovers and catches up with the rest of the system. In the next few months we will implement fuzzy replica production. In 1996 we will put effort in further enhancements of availability. We are developing a protocol for on-line non-blocking upgrade of dictionary data, a set of algebra methods for incremental on-line modification of data as a result of schema updates, and support for upgrade of basic software and disk structures.

Acknowledgements

We would like to thank Oddvar Risnes and César Galindo-Legaria for their constructive comments to the paper.

References

[Ahn94] Ilsoo Ahn. Database issues in telecommunications network management. In *Proceedings of ACM/SIGMOD (Management of Data)*, May 1994.

[Cri90] Flaviu Cristian. Understanding fault-tolerant distributed systems. Research report, IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120-6099, USA, 1990.

[ESC88] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In M. Stonebraker, editor, *Readings in Database Systems*, pages 259–273. Morgan Kaufmann Publishers, Inc., San Mateo, California, 1988.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, 1070 p., 1992.

[Gra91] Jim Gray, editor. *The Benchmark Handbook for database and transaction processing systems.* Morgan Kaufmann Publishers, 334 p., 1991.

[HLNW94] Michael Heytens, Sheralyn Listgarten, Marie-Anne Neimat, and Kevin Wilkinson. Smallbase: A main-memory dbms for high-performance applications (release 3.1), March 1994. Hewlett-Packard Laboratories, Palo Alto, CA, USA.

[HST+91] Svein-Olaf Hvasshovd, Tore Sæter, Øystein Torbjørnsen, Petter Moe, and Oddvar Risnes. A continously available and highly scalable transaction server: Design experience from the HypRa project. In *Proceedings of the 4th International Workshop on High Performance Transaction Systems*, September 1991.

[Hva92] Svein-Olaf Hvasshovd. *HypRa/TR: A Tuple Oriented Recovery Method for a Continuously Available Distributed DBMS on a Shared Nothing Multi-Computer.* PhD thesis, The Norwegian Institute of Technology, University of Trondheim, July 1992. 266 p. ISBN 82-7119-373-2. Also SINTEF DELAB Technical report STF40 A93085.

[JLRS94] H. V. Jagadish, Daniel Lieuwen, Rajeev Rastogi, and Avi Silberschatz. Dalí: A high performance main memory storage manager. In *Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile (VLDB '94)*, pages 48–59, September 1994.

[MHL+92] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead locking. *ACM Transactions on Database Systems*, 17(1):94–162, March 1992.

[Tik92] M. Tikkanen. TDMS: An embedded raeltime main-memory database management system. *Proceedings of Embedded and Real-Time Systems, The Finnish Aritificial Intelligence Society*, November 1992.

[Tik93] M. Tikkanen. Objects in a telecommunications oriented database. In *Proceedings of the Conceptual Modelling and Object-Oriented Programming Symposium*, November 1993.

[Tor95] Øystein Torbjørnsen. *Multi-Site Declustering Strategies for Very High Database Service Availability.* PhD thesis, The Norwegian Institute of Technology, University of Trondheim, January 1995. 186 p., ISBN 82-7119-759-2.