A real-time database for future telecommunication services

T. Niklander, J. Kiviniemi and K. Raatikainen
University of Helsinki, Department of Computer Science
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland.
Telephone: +358 9 7084 {4235,4427,4243}.
Fax: +358 9 7084 4441. E-mail: {tiina.niklander, jukka.kiviniemi,kimmo.raatikainen}@cs.Helsinki.FI

Abstract

Future telecommunication services will extensively exploit database technology. The persistent and temporal information needed in operations and management of the telecommunication networks and services will be in databases. The current Intelligent Network (IN) Recommendations of ITU-T imply that real-time transaction processing capabilities should be provided. Telecommunications Management Network (TMN) and Telecommunications Information Networking Architecture (TINA) are object oriented. An ideal database system supporting various telecommunication applications should be a fault-tolerant distributed real-time object-oriented database system.

The most challenging issue in designing a real-time transaction processing system for telecommunications is the handling of transactions in three categories having very different characteristics. A telecommunications database system should be able to support short but voluminous simple read transactions, long but voluminous simple updating transactions, and a few very long complex updating transactions in the same real-time database system.

Keywords

Real-time transactions, fault-tolerance, concurrency control and scheduling

1 INTRODUCTION

Databases will already in the near future have a central role in telecommunications networks. The information needed in operations and management of the nets will be collected into a logically uniform database. The world-wide nature of telecommunications prescribes that the only possibility to obtain the logical uniformity is the co-operation of autonomous databases.

In the research projects Darfin and RODAIN we have examined database needs in telecommunications. Based on our results and experience the most important features will be the real-time and object-orientation. Real-time transactions are really needed in telecommunications. In telecommunications we will need both soft transactions that may continue their execution after deadline but with a reduced lower priority and firm transactions that are terminated when their deadline expires. We do not believe that hard transactions will be used in near future because systems supporting hard transactions are too expensive for open telecommunication markets.

The RODAIN database architecture is a real-time, object-oriented, fault-tolerant, and distributed database management system. The RODAIN is designed to fulfil the requirements of a telecommunications database systems. The requirements are derived from the most important telecommunications standards including Intelligent Network (IN), Telecommunications Management Network (TMN), and Telecommunication Information Networking Architecture (TINA). The requirements of the telecommunications database architectures originate in the following areas: real-time access to data, fault tolerance, distribution, object orientation, efficiency, flexibility, multiple interfaces, and compatibility (Raatikainen 1997; Taina *et al.* 1996b, 1997).

The most challenging issue in designing a real-time transaction processing system for telecommunications is the handling of transactions in three categories having very different characteristics. A telecommunications database system should be able to support short but voluminous simple read transactions, long but voluminous simple updating transactions, and a few very long complex updating transactions in the same real-time database system. The RODAIN Database (Taina *et al.* 1996b) is designed to meet those constraints. In RODAIN the object-oriented approach was chosen because a special purpose object model for real-time transactions (Kiviniemi *et al.* 1996) can be used to provide the information needed in the concurrency control and real-time scheduling of heterogeneous transactions.

The rest of the paper is organised as follows. In Section 2 we briefly summarise the RODAIN database architecture. The process structure of the RODAIN Database Management System is presented in Section 3. In Section 4 we describe how transactions are processed in the RODAIN Database. In Section 5 we discuss some aspects of fault-tolerance in the RODAIN Database.

2 RODAIN DATABASE ARCHITECTURE

The RODAIN database architecture is a real-time, object-oriented, distributed, and fault-tolerant architecture of a database management system designed for telecommunications. Below we briefly summarise the essentials of the architecture while a detailed description can be found in Taina *et al.* (1996a-c).

A RODAIN Database consists of a set of autonomous RODAIN Database Nodes that interact with each other. Each database node may communicate with one or more applications, and an application may communicate with one or more database nodes. A RODAIN Database Node consists of Database Primary Node, Database Mirror Node, and Reliable Secondary Storage Subsystem (Figure 1). The primary and mirror node are identical having a set of subsystems:

- User Request Interpreter Subsystem (URIS),
- Distributed Database Subsystem (DDBS),
- Fault-Tolerance and Recovery Subsystem (FTRS),
- Watchdog Subsystem (WDS), and
- Object-Oriented Database Management Subsystem (OO-DBMS).

User Request Interpreter Subsystem

A database management system to be used in telecommunications must support several interfaces to the database including CS-1/CS-2 INAP, CMIP, CORBA, TINA. User Request Interpreter Subsystems translate various interfaces into a common connection language. Each URIS takes care of one specific interface. The URIS on the Primary Node is active. On the Mirror Node the URIS is passive.

Distributed Database Subsystem

A RODAIN Database Node may either be used as a stand-alone system or in co-operation with the other autonomous RODAIN Database Nodes. The database co-operation management in the Database Primary Node is left to the Distributed Database Subsystem. The Distributed Database Subsystem on Mirror Node is passive.

Fault-Tolerance and Recovery Subsystem

The subsystem controls communication between the Database Primary Node and the Database Mirror Node. It also co-operates with the Watchdog Subsystem to support fault-tolerance.

The FTRS on Primary Node handles transaction logs and recovery data. It takes care of saving transaction logs either into the Mirror Node or into the Secondary Storage Subsystem. Since the database only contains committed data, the FTRS only needs to handle redo logs.

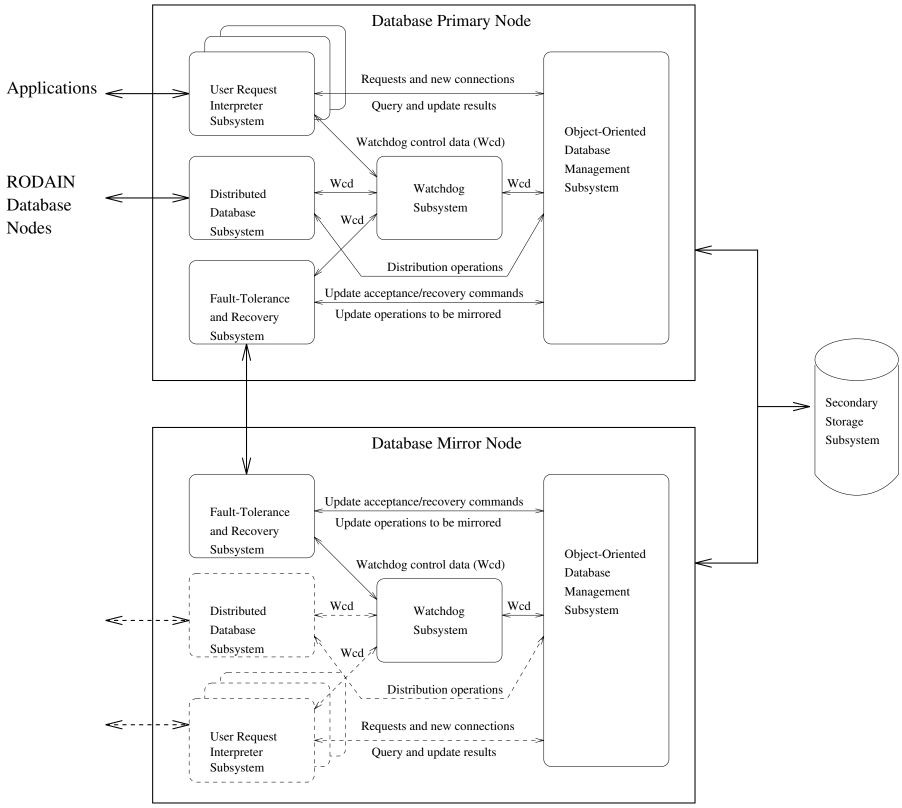

Figure 1 RODAIN database node.

The FTRS on the Mirror Node receives the logs from the FTRS on the Primary Node. It saves the logs into the Secondary Storage Subsystem and sends the corresponding update instructions to the OO-DBMS on the Mirror Node. The FTRS on the Mirror Node also receives the recovery data from the Primary Node and passes it to the local OO-DBMS.

Watchdog Subsystem
The subsystem watches over the other running subsystems locally on both Primary and Mirror Node. It communicates with the other subsystems to state the current node status. Upon a failure it restores the node. The Watchdog Subsystem notices when a system is down and starts a recovery process.

Object-Oriented Database Management Subsystem
The OO-DBMS is the main component of a RODAIN Database Node both on Primary Node and on Mirror Node. It maintains databases, real-time constraints, integrity, and concurrency control. It offers object storing, querying, and

transaction services for the URISes and the DDBS. The OO-DBMS consists of a set of database processes that use database services to resolve requests from other subsystems and of a set of manager services that implement database functionality.

3 OBJECT-ORIENTED DATABASE MANAGEMENT SUBSYSTEM IN RODAIN

The Object-Oriented Database Management Subsystem (OO-DBMS) implements the functionality of the RODAIN database. It offers object storing, querying, and transaction services for User Request Interpreter Subsystem, which communicates with an application outside RODAIN Database Node.

Transactions are executed in the Database Primary Node. The Database Mirror Node does not accept incoming transactions. When running as the Database Primary Node the node is either in the normal primary mode or in the transient mode. The key difference between the two modes is in the way how the transaction log is handled.

Processes in a RODAIN Node are induced from different sources. The sources include operating system, network communication system, database processes, and executing transactions. Processes induced by the RODAIN Database system are divided into priority levels. Database services other than transactions have usually a fixed priority. The Transaction Processes have a priority area, in which their priorities can vary. The priority of a Transaction Process depends on the deadline, importance, and other properties of the transaction. The priority can (and usually does) change during the runtime of the transaction.

The processes in the Database Primary Node are depicted in Figure 2 using a formalism of an extended DARTS software design method (Gomaa 1984) for real-time system. Below we briefly summarise the basic concepts of processes that are relevant when transactions are processed.

Runtime Transaction Controller

The Runtime Transaction Controller (RTC) accepts new transaction requests from applications. The RTC allocates one Transaction Process from the pool of Transaction Processes to serve the incoming transaction. Based on the attribute values of the transaction instance the RTC assigns a priority to the Transaction Process. The Runtime Transaction Controller can deny an incoming transaction request in an overload situation.

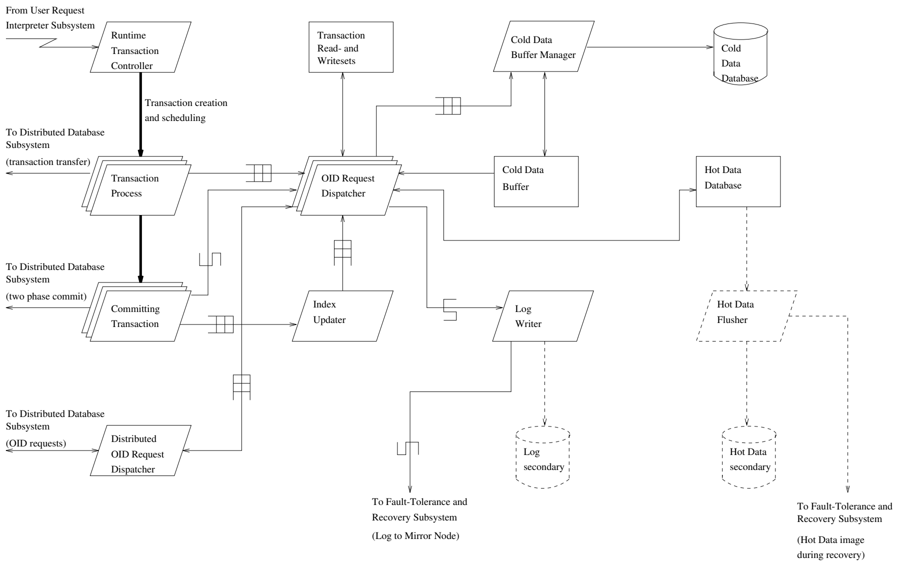

Figure 2 Processes in RODAIN Database Primary Node.

The Runtime Transaction Controller is also responsible for participating in transaction scheduling. Transaction scheduling is done by modifying the priority of each Transaction Process. The RTC adjusts the priorities of each transaction based on the selected scheduling policy. When a transaction is aborted and restarted due to concurrency control policy, the transaction is restarted in the same Transaction Process.

The Runtime Transaction Controller also handles transactions that have missed their deadlines. According to the over-deadline handling scheme, which is a mandatory attribute of any real-time transaction in the RODAIN data model (Kiviniemi *et al.* 1996), the RTC either aborts the transaction or lets it run with a lowered priority and importance. In both cases, the originator of the transaction is notified about the missed deadline.

Transaction Processes

A Transaction Process is invoked to handle requests sent by an application. A request is either a request to access an object or an invocation of method in an object (method call). Transaction Processes are permanent processes that are allocated to execute transactions. When a transaction is completed, the Transaction Process that carried out the execution of the transaction is returned into the pool of free Transaction Processes. When a transaction is aborted and restarted due to concurrency control, the same Transaction Process is used.

A Transaction Process offers the functionality specified in the RODAIN data model as well as query optimisation and usage of indices in queries. The object methods are executed in the memory space of the Transaction Process.

OID Request Dispatchers

An OID Request Dispatcher is a process that receives OID read and write requests from Transaction Processes and executes them. When a Transaction Process performs a request to access an object, the request is placed into the request queue. One OID Request Dispatcher process gets the request from the queue. When a request is completed, a result message is sent back through a buffered communication channel. Therefore, the requesting Transaction Process can decide, whether it waits for the completion of one request or whether it sends multiple requests into the queue before waiting a completion.

An OID Request Dispatcher offers services for reading and (pre)writing an object, and for validating and committing a transaction. A data accessing method depends on whether the requested object belongs to the hot data or to the cold data. When the accessed object is in the hot data, the OID Request Dispatcher computes its direct physical address in the main memory database and performs the operation. When the accessed object is in the cold data, the OID Request Dispatcher first tries to access data in the cold data buffer. If the object is not in the buffer, the request is forwarded to the Cold Data Buffer Manager. When an object is accessed, the readset or writeset of the requesting transaction is updated.

All OID Request Dispatcher processes are installed during the startup of the database. The number of processes is specified as an initialisation parameter of the RODAIN Database.

Committing Transactions

A Committing Transaction is a Transaction Process that has entered the commit phase. The Committing Transactions have the same functionality as Transaction Processes but each Committing Transaction has a higher priority than any Transaction Process. A Committing Transaction performs the validation of the transaction commit using the readset and the writeset of the transaction. If the validation is successful, all modified objects are written into the database and the transaction is committed. If the concurrency control detects a conflict, at least one transaction is aborted and restarted.

Cold Data Buffer Manager

The Cold Data Buffer Manager (CDBM) receives read and write requests to the cold data from the OID Request Dispatcher. Since a read request sent to the CDBM can not be resolved from the Cold Data Buffer, the request is resolved from the Secondary Storage Subsystem. A write request causes the written object to be pinned into the Cold Data Buffer. In the case of transaction commit, the modified objects are written into the Secondary Storage Subsystem and unpinned before the commit is accepted.

When the Cold Data Buffer is full, the CDBM makes free space by removing some objects from the buffer. The pinned objects are the last ones to be removed, since they need to be written into a temporary disk file. When ever possible the objects to be removed are selected from the set of unpinned objects. A priority-LRU method (Carey *et al.* 1989) is used in the selection.

Log Writer

The Log Writer handles log write commands. When the Database Primary Node is in the normal primary mode, the requests are passed to the Mirror Node. When the Database Primary Node is in the transient mode, the log write requests are forwarded to the Secondary Storage Subsystem. In both cases the write process is synchronous. Thus, a log write operation is finished when it is guaranteed that the entry is permanently stored either into the Mirror Node or into the Secondary Storage Subsystem.

Hot Data Flusher

The Hot Data Flusher writes the contents of the main memory database into the Secondary Storage Subsystem. This process is used in the Primary Node only when the node is in the transient mode.

4 TRANSACTION PROCESSING IN RODAIN

As mentioned in the Introduction, a database used in telecommunications meets three different types of transactions. Firstly, the short queries are used to retrieve information about characteristics of the called subscriber and the calling telephone line. These transactions have real-time requirements but strict serializability is usually not needed. Secondly, a telecommunications database is often a subject to transactions that update information of a single subscriber. Thirdly, the large TMN transactions are used to update large number of records. These transactions do not have real-time requirements but they have strict atomicity and isolation requirements. The challenge is that all the three transaction types must be scheduled in a single database but the scheduling requirements are conflicting. Therefore, we have paid special attention to overcome transaction scheduling problems in the RODAIN database system.

In RODAIN the scheduling and concurrency control exploits the real-time properties available in the RODAIN data model. Below we highlight the essentials in the RODAIN data model, transaction scheduling and concurrency control, and in overload management. All these aspects are key issues in developing a real-time database system (Yu *et al*. 1994).

4.1 Real-time properties in RODAIN data model

In the RODAIN data model each transaction has the properties **`deadline`** and **`deadline_type`**. The **`deadline`** property indicates the time before which the transaction should commit. The granularity and time model for deadlines are not specified in the data model. The **`deadline_type`** property specifies how a transaction is treated when the deadline expires. In RODAIN the alternatives of the treatment are firm, soft, and none.

When a transaction is started, it is timestamped with the start time of the transaction. An application can also specify the **`expected_execution_time`** property for the transaction. If the value of **`expected_execution_time`** is not specified in the transaction, then a weighted average of past instances is used. The granularity and time model in the **`expected_execution_time`** property is the same as in the **`deadline`** property. The **`expected_execution_time`** property can be changed during the execution.

In scheduling the following properties are also taken into account:

- **`transaction_importance`**,
- **`deadline_expiration_action`**, and
- **`late_transaction_importance`**.

The **`transaction_importance`** property specifies the relative importance of the transaction in respect to other transactions in the system. The **`deadline_expiration_action`** property specifies which actions are to be taken when the deadline expires. This property is currently an implicit one because its value depends on the value of the **`deadline_type`** property. Possible values are abort and continue. If the value is abort, the transaction is terminated when the deadline expires. If the value is continue, the execution is continued with a lowered priority (see Kao *et al.* 1995). The new priority value is derived from the value of the **`late_transaction_importance`** property.

The goal of real-time scheduling is to maximize the number of transactions that complete before their deadlines. However, this is not always possible. In these overload situations transactions having low values of the **`transaction_importance`** property are sacrificed in favor of those having high values. This is the concept of overload resolution policy used in the RODAIN database. The RODAIN database has also an overload prevention policy that controls the creation of new instances of transactions.

In the RODAIN model transactions have the **`isolation_level`** property that defines the correctness criterion sufficient for the transaction. This criterion specifies which level of correctness is needed when the transaction is executed. The isolation level of transaction is compared to the isolation level of accessed objects and the stringiest level is used. Since each transaction is individually treated, every transaction accessing the same object will receive at least the specified correctness level.

The **transaction_behavior** property indicates how a transaction is going to access data in the database. The RODAIN database model introduces all the three transaction behaviours described in Ramamritham (1993):

1. A read-only transaction reads data from a database and does not make any updates.
2. A write-only transaction collects data from the outer world and writes it into the database.
3. A read-write transaction reads data from a database, updates it, and stores it back to a database.

The **transaction_behavior** property guarantees that operations violating the behaviour will raise an exception that aborts the transaction. The default behavior of transaction is a read-write transaction.

4.2 Basic principles of scheduling and concurrency control

In our prototype implementation we are experimenting with a two level scheduling algorithm. The upper level is used to provide a reasonable fairness between different types of transactions. The lower level is used to maximise the number of transactions completed in time.

When a transaction arrives into the system, it is assigned into one of the following three groups: 1) short real-time transactions, 2) long real-time transactions, and 3) non real-time transactions. The groups correspond to the three transaction types common in telecommunications. The assignment is based on the following three properties: **deadline_type**, **expected_execution_time**, and **transaction_behavior**. Each group gets a predefined fraction (system parameter) of available cpu time in a round robin fashion. Actually, we have different fractions for low, medium, and high system load.

In the group of non real-time transactions the scheduling policy is round robin with a relatively long time slice. In the two groups of real-time transactions we use a weighted least slack time first scheduling. The weight is based on the **deadline_type** and **transaction_importance** properties. In the group of short real-time transactions the scheduling is non pre-emptive. In the group of long real-time transactions the scheduling is pre-emptive in the sense that when the priority order of two transactions changes, the running transaction is switched. The scheduler calculates the switch time beforehand and uses a threshold to prevent the so called "continuous process switch" phenomenon.

As the concurrency control algorithm we use our own modification of the OCC-TI algorithm (Son *et al.*1992, Lee *et al.* 1993). The modifications take into account the correction criterion (the **isolation_level** property), the **transaction_importance** property, and the remaining slack time of conflicting transactions. In concurrency control our main worry is read-write conflicts because write-write conflicts are rare in telecommunication applications.

We use two basic techniques that are derived from the application semantics in telecommunication. The massive updates, which are primarily used in service management, are carried out as non real-time transactions. They create new versions of the objects that become valid in the future. Features of active databases supported in RODAIN are tailored to trigger massive new current version updates. The read-write conflicts between the simple queries and updates are removed by using the so called τ-serializability introduced in Raatikainen *et al.* (1995) as the correctness criterion. The principal idea in the τ-serializability is to allow transactions to read (slightly) old data.

Suppose that the transaction *a* has read the object **x** at time t_a and that the transaction *b* has committed an update of **x** at time t_b, $t_a < t_b$. Further, suppose that *a* tries to commit at time t_c, $t_b < t_c$. In the τ-serializability the validation of *a* fails only if $t_a+\min(\tau_{\mathbf{x}},\tau_a) < t_b$ or $t_b+\min(\tau_{\mathbf{x}},\tau_b) < t_c$, where τ_a, τ_b, and $\tau_{\mathbf{x}}$ are the τ-values of transaction *a*, transaction *b*, and object **x**, respectively. The τ-value specifies how long a time an old value is useful after an update.

4.3 Overload Management

The overload policy defines how the RODAIN Database Primary Node behaves when its load temporarily exceeds its processing capacity. In an overload situation all transactions can not meet their deadlines. Thus, the system must make some decisions either to prevent overloads beforehand or to resolve overload situation when it occurs.

An overload situation can be prevented by controlling the number of executing Transaction Processes and OID Request Dispatchers. In the RODAIN Database the Runtime Transaction Controller can deny new transaction requests. In this case the request originator is notified. Another way used in the RODAIN to prevent overload situations is the control for the number of running OID Request Dispatchers. In this case transactions are accepted but their execution is delayed due to queuing OID Request Dispatcher resources. The number of Transaction Processes and the number of OID Request Dispatchers are system-wide parameters used in tuning the database performance.

In addition to the preventive overload control the RODAIN Database also exploits an observative overload policy in overload situations. When too many transactions have missed their deadline during the monitoring interval, the system load is decreased by aborting transactions that impart lower value to the system.

5 FAULT-TOLERANCE IN RODAIN

The requirement of high availability has lead us to design RODAIN Database Node so that it remains, at least partially, functional even if only one functional node operates normally. If all hardware nodes have failed, then the systems availability requirement is also violated. However, the recovery from the Secondary Storage Subsystem (SSS) turns the database into a consistent state but a few committed transactions might be lost.

The RODAIN Database, consisting of two functional nodes (the Primary Node and the Mirror Node), can currently tolerate a failure. If the system had more Mirror Nodes, then it would remain functional even it all but one node fail.

When both nodes are functional, one node (Primary Node) is active and executes the transactions. The other node (Mirror Node) is passive. The mirror only maintains its own copy of main-memory database according to information received from the primary.

Currently our failure model is based on the assumption that the whole node fails. We also assume that the communication link between the Primary Node and the Mirror Node does not fail alone. In the RODAIN system this assumption is not unrealistic since the nodes are connected through two physically separate communication paths.

5.1 Operations when both nodes functional

The Primary Node executes all arriving transactions. It uses deferred writes with optimistic scheduling just to avoid operations due to aborted transactions. The rollback in the RODAIN Database is very simple: just to remove the data structures of aborted transaction. Some cleanup operations are needed in the data structures used by concurrency control but those operations are the same as after a transaction has successfully committed.

The fault-tolerance in RODAIN is achieved by sending transaction logs including the commit log to the Mirror Node. The idea of using separate node or processor for log handling is not new; see e.g. Levy *et al.* (1992) and Lehman *et al.* (1987). In a traditional database system the logs are stored in stable memory like disk before transaction is allowed to commit. In RODAIN we assume that the Mirror Node is stable. This means that a transaction may commit when its log has arrived into the Mirror Node.

The Mirror Node uses the log to update its own database. It also stores the log to reliable disk subsystem (Secondary Storage Subsystem, SSS). If both nodes fail at the same time, our fault-tolerance assumptions are violated. However, the logs stored on the SSS guarantee that almost all committed transactions have their data maintained in the RODAIN system.

In order to make the recovery fast, the Mirror Node also makes periodic copies of its database image into the SSS. This copy is used to reduce the number of log records needed to be processed when the failed node is recovering. Checkpointing is used to mark the creation of image in the log. Our checkpointing scheme is similar to the one presented in Jagadish *et al.* (1993).

Since we make the checkpoints from a copy of the database, it never contains any uncommitted data. Therefore, we do not need any undo operation in RODAIN.

5.2 Operations when one node fails

The operations when a node fails are slightly different depending on which one of the functional nodes failed. If the passive Mirror Node fails, then the active Primary Node has to start saving transaction logs directly into stable memory. This guarantees that a huge amount of committed data will not be lost if also the active Primary Node fails.

If the active Primary Node fails, the operations are little more complex because the Mirror Node has to change its functionality to new Primary Node. First of all, the mirror updates its own copy of the main-memory database using the unprocessed logs. At the same time all non-active subsystems are activated. After both steps have been completed, the mirror can start to act as the new active Primary Node.

Currently, we loose the active transactions during this role switch. However, the amount of lost transactions can be reduced if the User Request Interpreter Subsystems in the Mirror Node maintain list of uncompleted transactions. The new active node could restart transactions that are feasible and assumed to be able to finish before deadline. The co-operation between the URISes in the two nodes is the way how fault-tolerance can be further increased in RODAIN. We have decided not to support transaction migration since we assume that most failures are due to hardware (or operating system) problems.

5.3 Operations when only one node is functional

The only functional node is always an active Primary Node in transient mode. It executes transactions and saves logs directly to the SSS. This makes the transaction commit slower but the data is not totally lost if also this node fails. However, the commit procedure can be speed up by using the so called group commit mechanism.

If the failed node remains unavailable for a long period, the Primary Node may have to also make a full copy of its main-memory database to the SSS, just to make the recovery of the other node faster.

5.4 Operations when failed node recovers

The functional active Primary Node maintains its state in transient mode until the recovering new Mirror Node informs its presence. The Primary Node then changes its state back to normal mode.

The recovering node first loads the latest database image. Then it will process all the stored log records to update the database. When the recovering node is “up-to-date”, it informs the Primary Node that it is ready to serve as the Mirror Node. The Primary Node can then stop storing of logs into the SSS and start sending them to Mirror Node for further processing.

The time when the recovering node may inform its presence is not unique. The earliest possible point is when the recovering node has loaded the image and the latest possible one is when the recovering node has processed all the logs. It should, however, be noticed that there may always be new log records to be processed because the active Primary Node is writing new ones unless the system is idle. Therefore, the passive recovering Mirror Node should take the responsibility of log handling even if it has not yet processed all available log records.

5.5 Transaction commit in details

Transaction commit is the critical point in achieving fault-tolerance in database. When transaction commits, all of its logs are sent to the Mirror Node and stored to stable memory in the Secondary Storage Subsystem.

We need to consider only updating transactions since the log contains only redo operations with after images. Read-only transactions do not change the state of the database which implies that they do not produce log entries.

During the transaction commit phase the executing transaction process makes all the changes permanent into the database. The commit is allowed only if the transaction does not conflict with other transactions. The transaction commit is finished only when the logs are stored into the Mirror Node or directly into the SSS. Since the transactions are (τ-)serialised in their commit order, the Mirror Node can use the commit order as the order in which it updates its copy of the database. Therefore, the Mirror Node should use the simple “FIFO - one transaction at a time” scheduling when updating its own database.

The last log record of each transaction received by the mirror contains only the commit operation. This fact is used in synchronisation so that the OID Request Dispatcher waits for its processing. The Fault-Tolerance and Recovery Subsystem (FTRS) on the Primary Node sends to the FTRS on the Mirror Node all log records of a transaction. When the FTRS on the Mirror Node has received all the log records related to a transaction, which is indicated by the commit record, the FTRS on the Mirror Node sends an acknowledgement to the FTRS on the Primary Node.

The FTRS on the Primary Node does not destroy log records until the FTRS on the Mirror Node has acknowledged them. The FTRS on the Mirror Node sends this acknowledgement message only when it has saved the log into the SSS. This extra storing is used to avoid loosing logs of committed transactions in the case of a Mirror Node failure.

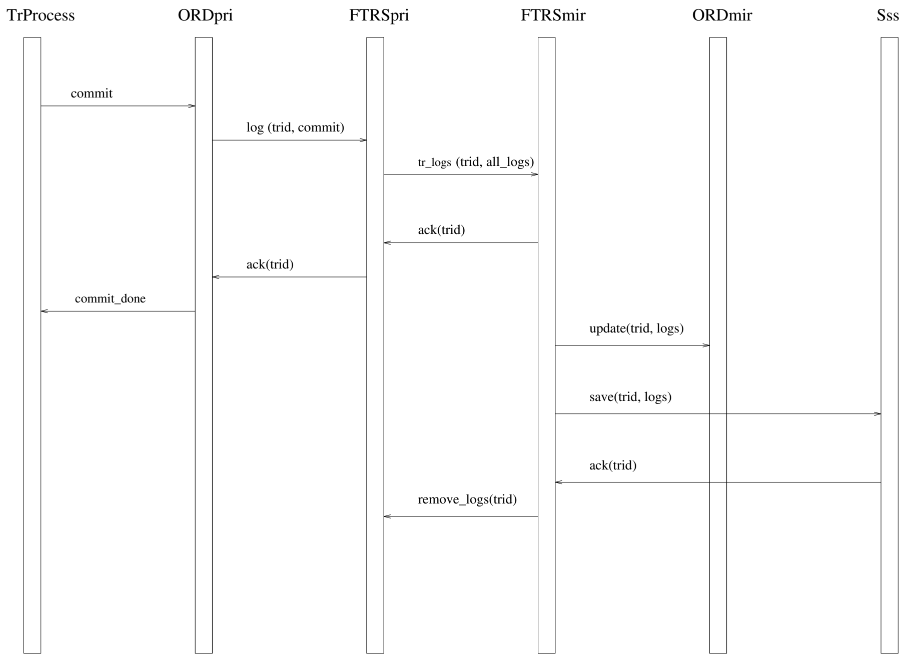

Figure 3 Message sequence chart in the commit phase

Concurrently with storing the log, the FTRS on the Mirror Node also updates its own copy of the database. The log records on the Mirror Node are removed when both the update and the storing has been done. The update and storing can be done in any order.

Should the Mirror Node fail, the FTRS on the Primary Node would save all log records that are still in its buffers. If the Primary Node fails, all the log records that are not yet sent to Mirror Node are lost. However, these records are from transactions that are not yet committed and, therefore, the changes can be lost without violating the consistency of the database.

Figure 3 shows the messages and their timings sent during the commit phase between the functional entities in the Primary and Mirror Node.

6 SUMMARY

We have described the RODAIN Database architecture and main parts of its prototype implementation. The RODAIN Database architecture is designed to meet the challenge of future telecommunication systems including Intelligent

Networks, Telecommunications Management Network, Telecommunication Information Networking Architecture.

In order to fulfil the requirements of the next generation telecommunications systems, the database architecture must be fault-tolerant and support real-time transactions with explicit deadlines. The internals of the RODAIN DBMS described are designed to meet the requirements of telecommunications applications including real-time access to data, fault tolerance, distribution, object orientation, efficiency, flexibility, multiple interfaces, and compatibility with telecommunications practices. The requirements are, in some extent, conflicting. Therefore, the RODAIN Database is based on trade-offs; novel and innovative solutions are used only when old and exercised methods are found to be insufficient.

In the RODAIN data model we defined real-time objects and real-time characteristics. The additional characteristics were designed so that the original ODMG-93 model (Cattell 1994) is a true subset of the extended model. The motivation for the extensions was to provide the real-time scheduler and concurrency controller enough knowledge to overcome the problems due to heterogeneous transaction lengths that are met in telecommunication applications. Availability of RODAIN is achieved through using a database mirror. The mirror is also used for log processing, which reduces load in the primary database node and shortens the commit times of transactions.

7 ACKNOWLEDGEMENTS

This work has been carried out in the research project RODAIN (1995-) funded by the Finnish Technology Development Center (TEKES) together with Nokia Telecommunications, Solid Information Technology, and Telecom Finland. The authors want to thank Pasi Porkka, Juha Taina, and Martti Tienari from the Department of Computer Science in the University of Helsinki for the fruitful discussions and valuable comments during the research. The industrial partners (Kyösti Laiho and Artturi Tarjanne from Solid, Jussi Ollikainen, Tapani Karttunen and Jari Vänttinen from Telecom Finland, Juha Lipiäinen, Simo Lähdesmäki and Petri Nuuttila from Nokia Telecommunications) have provided useful information and feedback comments during the project.

8 REFERENCES

Carey, M.J., Jauhari, R. and Livny, M. (1989) Priority in DBMS Resource Scheduling, in *Proceedings of the 15th Very Large DataBase Conference*, 397-410, Morgan Kaufmann, San Mateo, Calif.

Cattell, R.G.G. (ed.) (1994) *The Object Database Standard: ODMG-93*. Morgan Kauffmann, San Francisco, Calif.

Gomaa, H. (1984) A Software Design Methods for Real-Time Systems. *Communications of the ACM*, **27**, 9, 938-49.

Jagadish, H.V., Silberschatz, A. and Sudarshan, S. (1993) Recovering from Main-Memory Lapses, in *Proceedings of the 19th Very Large DataBase Conference*, 391–404, Morgan Kaufmann, San Mateo.

Kao, B. and Garcia-Molina, H. (1995) An Overview of Real-Time Database Systems, in *Advances in Real-Time Systems* (ed. S.H. Son), 463-86, Prentice-Hall.

Kiviniemi, J. and Raatikainen, K.E.E. (1996) *Object-Oriented Data Model for Telecommunications*. Technical Report C-1996-75, Department of Computer Science, University of Helsinki, Finland.

Lee, J. and Son, S.H. (1993) Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems, in *Proceedings of Real-Time Systems Symposium*, 66-75, IEEE Computer Society.

Lehman, T.J. and Carey, M.J. (1987) A Recovery Algorithm for A High-Performance Memory-Resident Database System, in *Proceedings of ACM SIGMOD 1987 Annual Conference*, 104–17, ACM SIGMOD.

Levy, E. and Silberschatz, A. (1992) Incremental Recovery in Main Memory Database Systems. *IEEE Transactions on Knowledge and Data Engineering*, **4**, 6, 529–40.

Porkka, P. and Raatikainen, K.E.E. (1997) CORBA access to telecommunications databases, in *Intelligent Networks and Intelligence in Networks* (ed. D. Gaiti), Chapman & Hall, London.

Raatikainen, K.E.E. (1997) Real-Time Dtabases in Telecommunications, in *Real-Time Database Systems: Issues and Applications* (ed. A. Bestavros *et al.*), Kluwer.

Raatikainen, K.E.E., Karttunen, T., Martikainen, O. and Taina, J. (1995) Evaluation of Database Architectures for Intelligent Networks, in *Proceeding of Telecom95 Technical Summit*, Vol. 2, 549-53, ITU, Geneva, Switzerland.

Ramamritham, K. (1993) Real-Time Databases. *Distributed and Parallel Databases*, **1**, 199-226.

Son, S.H., Lee, J.and Lee, Y. (1992) Hybrid Protocols Using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control. *The Journal of Real-Time Systems*, **4**, 2, 269-76.

Taina, J. and Raatikainen, K.E.E. (1996a) Design Issues and Experimental Database Architecture for Telecommunications, in *Intelligent Networks and New Technologies* (ed. J. Nørgaard and V.B. Iversen), 121–39, Chapman & Hall, London.

Taina, J. and Raatikainen, K.E.E. (1996b) Experimental real-time object-oriented database architecture for intelligent networks. *Journal of Engineering Intelligent Systems*, **4**, 3, 57–63.

Taina, J. and Raatikainen, K.E.E. (1996c) RODAIN: A Real-Time Object-Orinted Database System for Telecommunications, in *Proceedings of the*

DART'96 Workshop (ed. N. Soparkar and K. Ramamritham), 12–5, University of Massachusetts.

Taina, J. and Raatikainen, K.E.E. (1997) Database usage and requirements in intelligent networks, in *Intelligent Networks and Intelligence in Networks* (ed. D. Gaiti), Chapman & Hall, London.

Yu, P.S., Wu, K.-L. and Son, S.H. (1994) On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE*, **82**, 1, 140-57.

8 BIOGRAPHY

Tiina Niklander received her M.Sc. degree in computer science from the University of Helsinki, in 1993. She is currently a Ph.D. student in computer science at the University of Helsinki and a researcher in the RODAIN project. Her research interests include fault-tolerance and real-time databases.

Jukka Kiviniemi is completing his M.Sc. studies in computer science at the University of Helsinki. He is currently a research assistant in the RODAIN project. His research interests include real-time and distributed databases, and real-time systems.

Kimmo Raatikainen received the Ph.D. degree in computer science from the University of Helsinki, in 1990. He is currently an associate professor in computer science at the University of Helsinki. He is a member of ACM, IEEE (Communications and Computer Societies), and IFIP TC6 Special Interest Group of Intelligent Networks. His research interests include nomadic computing, telecommunications software architectures, and real-time databases.