Fault tolerance features of Rodain Database Architecture for Intelligent Networks

Tiina Niklander

Department of Computer Science, University of Helsinki

P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland

E-mail: Tiina.Niklander@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 kept in databases. The current Intelligent Network (IN) recommendations of ITU–T imply that real–time transaction processing capabilities should be provided. They also imply that the database should be fault tolerant, since the allowed down–time is only few seconds per failure.

In the research project Rodain the main objective is to design and specify a fault–tolerant real–time database architecture for telecommunications applications and to implement a prototype based on that architecture.

1 Introduction

New telecommunication services are less and less based on concrete hardware components. They are more based on the existing hardware behaving in a different way. For this change in the service creation a new model

is needed: the intelligent network concept [3]. It is a telecommunication network concept, that does not require large modifications to the existing hardware or software components when new services are created.

The knowledge in the intelligent network is mainly collected in the Service Data Function [4]. The actual switching networks accesses the Service Data Function (SDF) through Service Control Function every time it needs the new IN functionalities. A Service Data Function is actually a database system.

The Intelligent Network requirements say that the SDF should be able to process thousands of queries within one second. Although this requirement seems quite huge, it is still reasonable as we may have one database serving a number of telephone switches. Each switch can have hundreds, even tens of thousands, actual telephone lines connected to it. The temporal load can in fact climb very high.

The SDF must not only answer a huge number of queries, but it must answer them fast. A typical telephone user does not want to wait for a long time for the dial tone or for the actual call connection. He or she is used to it happening within few seconds, and the new services are not expected to take much longer than that.

The current telephone system is quite reliable, or at least it is a goal to achieve. The whole switching system is supposed not to be down more than few seconds per year. It is desirable that the new services of the intelligent network concept are available when ever the basic telecommunication network is. Therefore also the database system is allowed to be down only few seconds at a time.

The Rodain Database Architecture is designed to fulfill the requirements mentioned above. The Database architecture needs real–time features for timely answers. Main–memory database is the only possibility for timely critical data because some access requirements are so tight that the access from disk is not possible. Fault tolerance is the base for high availability.

In this paper we first present overview to our database architecture in Section 2. Services of the database system are presented in Section 3. Section 4 presents processes used in the functional nodes of Rodain Database. The fault tolerance features of the Rodain Database prototype are discussed in Section 5.

2 Overview of the Rodain Database Architecture

The Rodain DBMS architecture is a *real–time object–oriented database management system architecture*. It consists of a set of autonomous Rodain Database Nodes (Figure 1) that interact with each other. Each Database Node may communicate with one or more applications, and each application may communicate with one or more database nodes. Applications and database nodes may be geographically distributed.

To increase the database availability the Rodain Database Node consists of two identical co–operative nodes. One of the nodes is acting as the Database Primary Node and the other one is mirroring the Primary Node. When necessary the Primary and the Mirror Node can switch their roles. That is done when a failure occurs. When only one node is functional we call it a Transient Node. It acts as the Primary Node, but is not accompanied by a functional Mirror Node.

The whole Rodain Database is divided in two parts. Each data item may belong to one of the two heat groups: hot or cold data [10]. They are stored in different databases within the Rodain Database Node. Hot

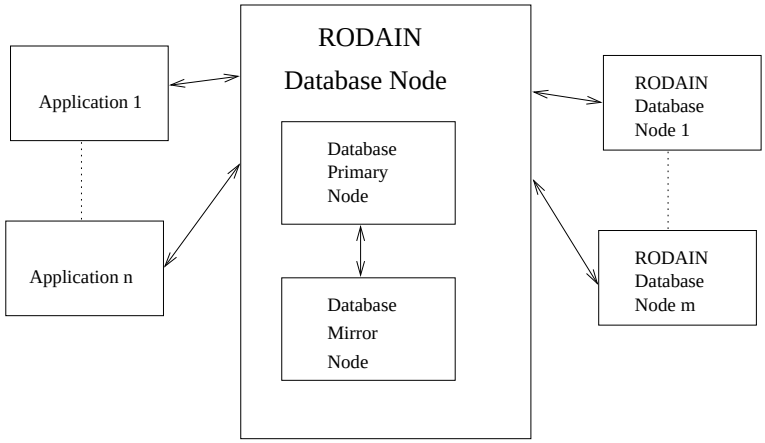

Figure 1. Overview of the Rodain Database Architecture

data is stored in a main memory database. All updates are done in main memory and a transaction log is maintained to keep the database in a consistent state. A secondary copy of hot data is located in the Mirror Node. Only a backup copy is maintained on the disk. Cold data is stored in a disk based database. Thus we use a hybrid data management method that is a combination of a main memory database and a disk based database.

Although the Rodain Database Nodes can co–operate and therefore support some forms of distribution, the database architecture is not a distributed database architecture. The Rodain Database architecture is based on the assumption that most of the transactions need to access local data on one autonomous node. We also assume that there is no replication between Rodain Database Nodes. The data has only one primary copy, but there can be several cached copies of it.

Each Rodain Database Node consists of Database Primary Node, Database Mirror Node and a reliable Secondary Storage Subsystem (Figure 2). The Primary and Mirror Node are identical and they can be switched. Both nodes have a set of subsystems that communicate with each other. The subsystems are: User Request Interpreter Subsystem (URIS), Distributed Database Subsystem (DDS), Fault–Tolerance and Recovery Subsystem (FTRS), Watchdog Subsystem (WS), and Object–Oriented Database Management Subsystem (OO–DBMS).

The Secondary Storage Subsystem (SSS) is a shared disk storage accessed by both the Primary and the Mirror Node. It is used for permanently storing cold data database, copies of hot data database, and log information.

User Request Interpreter Subsystem. The Rodain Database Node can have multiple application interfaces. Each interface is handled by one specific User Request Interpreter Subsystem. It translates its own interface language into a common connection language that the database management subsystem understands. The URISes on the Primary Node are active, because the clients communicate only with the Primary Node. On the Mirror Node the URISes are passive or do not exist.

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

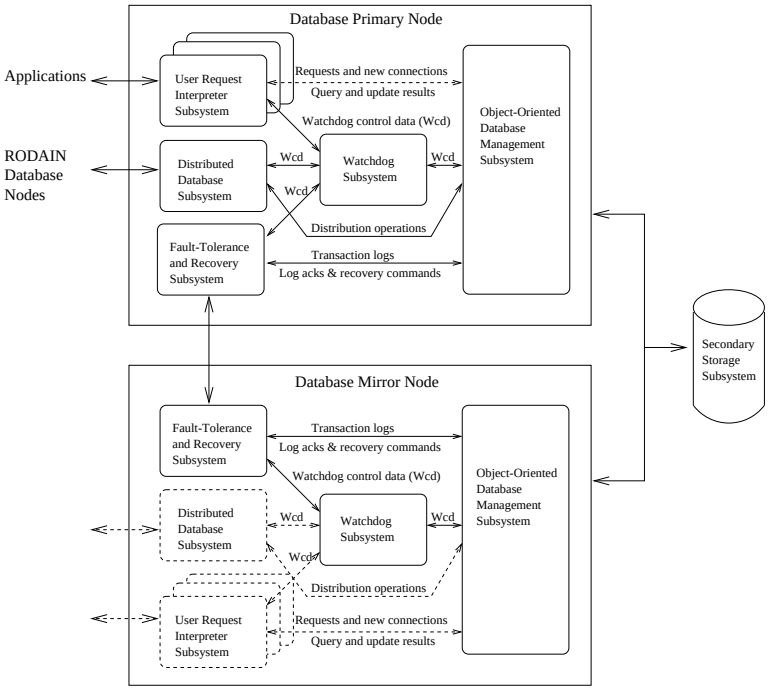

Figure 2. Rodain Database Node

Subsystem. The Distributed Database Subsystem on Mirror Node is passive or non–existent. It is activated when the Mirror Node becomes a new Primary or Transient Node.

Fault–Tolerance and Recovery Subsystem. This 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 the Primary Node handles transaction logs and failure information. It sends transaction logs to the Mirror Node. It also must notice when the Mirror Node stops functioning normally and report this to the Watchdog Subsystem for switching the node to the Transient Node.

On the Transient Node FTRS stores the logs directly to the disk on SSS.

The FTRS on the Mirror Node receives the logs sent by the Primary Node's FTRS. It then saves the logs to disk on SSS and gives needed update instruction's to the Mirror Node's Database Management Subsystem. When it notices that the Primary Node has failed, it informs the local Watchdog Subsystem.

Watchdog Subsystem. The Watchdog subsystem watches over the other local running subsystems both on the Primary and on the Mirror Node. Upon a failure it recovers the node.

On Primary Node when the local FTRS reports the failure of Mirror Node, the WS controls the node change to the Transient Node. This change affects mostly the FTRS, that must start storing the logs to the disk. On Mirror Node the failure of Primary Node generates more work. The WS must activate passive subsystems such as URIS and DDS. The FTRS must change its functionality from receiving logs to saving them to the disk on SSS.

Object–Oriented Database Management Subsystem. This is the main subsystem both on Primary Node and on Mirror Node. It maintains both hot and cold databases. It maintains real–time constraints of transactions, database integrity, and concurrency control. It consists of a set of database processes, that use database services to resolve requests from other subsystems, and a set of manager services that implement database functionality. The Object–Oriented Database Management Subsystem needs the Distributed Database Subsystem, when it can not solve an object request on the local database.

3 Database Manager Services and Service Layers

The Database Management Subsystem is divided into five layers (see Figure 3). A more detailed version is presented in [12]. The layers are *Transaction Execution Layer*, that is the visible layer to application, *Database Interface Layer* that is the visible layer to common database processes, *Object Layer* that handles the physical storage structures of objects, *Global Entity Layer* that is the visible interface layer to distribution and repli-

cation processing in the OO–DBMS, and *Real–time Core Layer* that implements low level database functionality such as real–time and physical data control.

Transaction Execution Layer provides the interface between application and database services. Every incoming request is handled as a transaction. Transactions are units of atomicity thus modifications done by a transaction are either committed or aborted as a whole. Transactions are scheduled according to their type and deadline. The goal of real–time transaction scheduling is to maximize the number of transactions, that will successfully finish before their deadlines [11].

Runtime Transaction Controlling Service accepts new transaction requests and redirects them to *Transaction Processing Service*. Runtime Transaction Controlling Service also performs transaction scheduling and overload management. Transaction Processing Service provides for transactions upper level services such as transaction committing and aborting. It also communicates with lower layers and passes results to the calling application. *Schema Manager Service* provides maintenance and management functions for type metadata.

Database Interface Layer is the interface between transactions and database services. It also offers higher level access to the database as, for example, attribute and relationship referencing, index handling, querying by values of an attribute, and query optimization. A full list can be found in [6]. The complete knowledge of the object model is visible up to this layer. Requests to lower layers are done with the OID of the accessed object.

Objects are accessed with an *Object Manager Service*, which provides functions for object fetching and storing, accessing object attributes, methods, relationships etc. Accessing of object instances can be done either in a sequential manner or with indexes assigned to attributes. *Index manager service* offers services for index creation, management, and query optimization by index values.

Object layer offers the *Physical Object Manager Service*, and the *Concurrency Controller Service*. These services are the core manager service for objects. On this layer the objects are all alike and the actual object model is no longer visible. Object accessing is always done with an OID of the object.

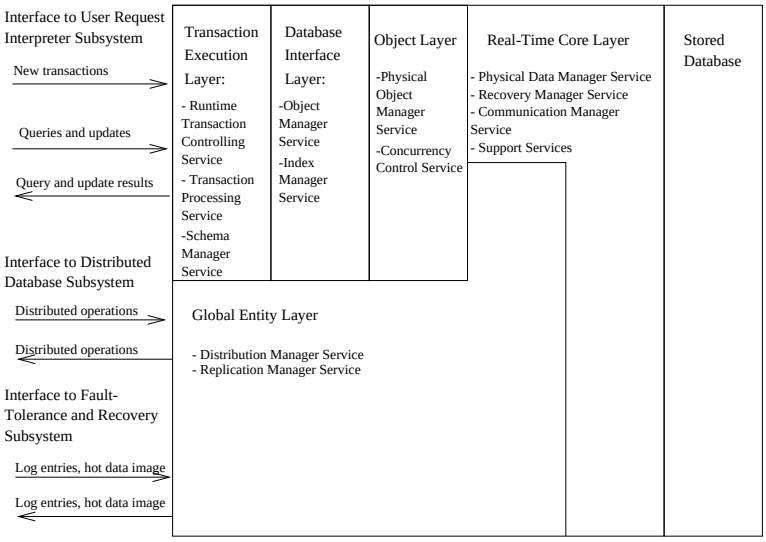

Figure 3. Layers of Database Management Subsystem

Physical Object Manager Service offers services for object creation, object deletion, and object accessing. Physical Object Manager Service maps OIDs to physical addresses in the stored database. The physical address depends on whether the object accessed is hot or cold, or resides in another database.

Physical Object Manager Service uses the Global Entity Layer when it cannot resolve an object request on the local database. It uses the Real–Time Core Layer services for object storing, fetching, and recovery.

Concurrency Control Service allows transactions to run in parallel. Data concurrency control in Rodain Database Node is based on optimistic concurrency control methods as presented in [13]. Concurrency Control Service also performs validation of local transactions. As a result of validation the transaction is normally committed, but it can be aborted.

Global Entity Layer offers services to all other layers and to special controller processes in order to support distribution and replication in the OO–DBMS. It consists of *Distribution Manager Service* and *Replication Manager Service*.

The Distribution Manager Service is a bidirectional service. When another service in the local database wants to access remote database, requests are handled by the Distribution Manager Service. The service redirects the request to the other database, which has also the Distribution Manager Service as a counterpart. The Replication Manager Service is used to send transaction logs to Mirror Node.

Real–time Core Layer implements the core operations of Database Management Subsystem. This lowest layer consists of *Physical Data Manager Service*, *Recovery Manager Service*, *Communication Manager Service*, and *Support Services*. The Communication Manager Service and Support Services are used by all other layers.

The Physical Data Manager Service is the only way to access the physical data stored in the database. It offers services for data storage, retrieval, and access estimates. Requests for this service are done with an object's address and length. The request may also contain the importance of the requesting transaction.

The Recovery Manager Service is the first service to gain control on Recovering Mirror Node when the Watchdog Subsystem has started OO–DBMS recovery operations. It returns database into a consistent state and restarts the database processes.

The Communication Manager Service offers services to the other managers to maintain communication channels to the other subsystems.

4 Processes in the Database Management Subsystem

In the RODAIN Database Node only Primary Node servers application requests. Therefore, all transactions are executed on the Primary Node. The Mirror Node does not accept connections from the applications. It starts transaction processing and accepts application's requests when it becomes a Transient Node (and later on the Primary Node). The active transactions are lost when the Primary Node fails. They are not migrated

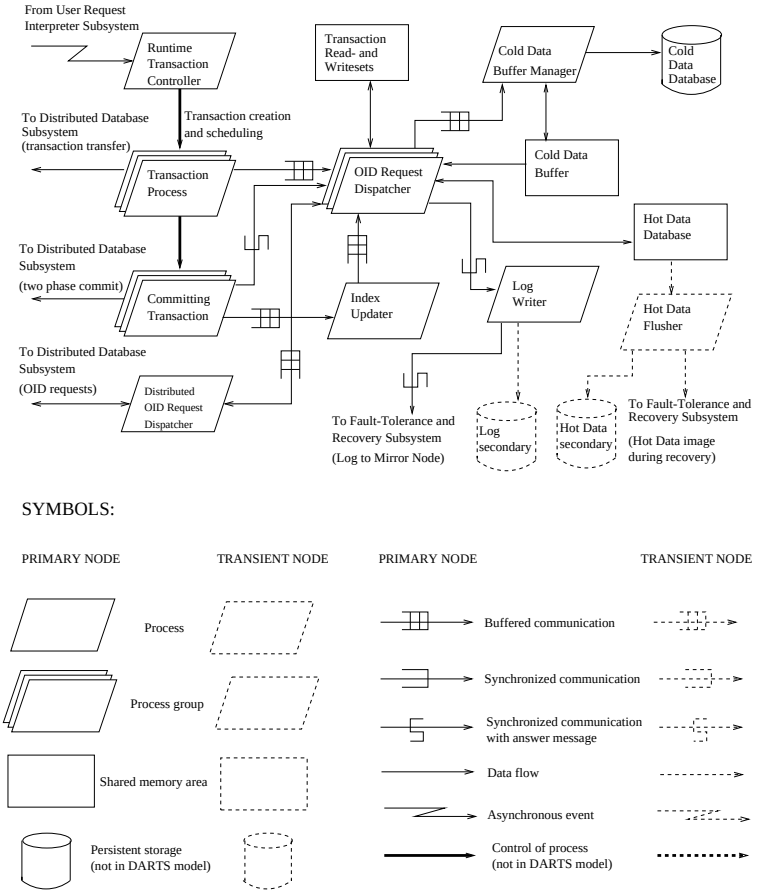

Figure 4. Processes in the Primary Node

to the new Transient Node. The applications notice the failure of Primary Node when they receive no response within given time frame.

The processes in the Primary Node are presented in Figure 4. The formalism in the figure is based on the DARTS software design method for real–time system [2] with some additions. Transient Node needs all Primary Node processes and operations with some extra operations, such

as hot data flushing and log writing directly to the disk. These Transient Node specific operations are marked with dashed lines. The Mirror Node uses a subset of the operations needed in the Transient Node.

Runtime Transaction Controller accepts new transaction requests from URIS, that is connected to the applications. Runtime Transaction Controller creates a new *Transaction Process* for each incoming transaction and assigns appropriate properties to it. These properties include a deadline and a transaction type. The Runtime Transaction Controller can deny an incoming transaction request in overload situation. Runtime Transaction Controller also handles transaction scheduling by adjusting the priorities of each transaction based on selected scheduling policy. The priorities are then used by the operating system for process scheduling.

Transaction Process is started to handle requests coming from an application. These requests can be either a single request that invokes a prespecified transaction method or a queue of object method calls. Transactions are transient processes which are created at the point of transaction start and killed when transaction terminates. When a transaction is restarted due to concurrency control, the process instance is not re–created. Instead the process executes the same transaction again.

OID Request Dispatchers offer services for object reading and writing, transaction validating and committing. Note that OID Request Dispatchers validate and commit only local transactions, thus the responsibility of distributed committing is left to the transaction processes.

The OID Request Dispatchers have one common request queue. They serve the arriving requests in the priority order. The ORDs are identical and each one can serve any arriving request. The object access is based on the object's OID only. For example, when a Transaction Process asks for an object to be read, it sends the object OID accompanied with the read command to the request queue. One of the ORDs gets the request from the queue and executes it. The result message containing full or partial object is sent to the requesting Transaction Process via a buffered communication channel.

A data accessing method depends on where the object is physically stored. When an accessed data is in the hot database, the OID Request Dispatcher computes direct physical address to the hot data database and performs the requested operation. When accessing cold data in the cold

database, it first tries to access data in cold data buffer and if it is not there, the request is forwarded to Cold Data Buffer Manager. In the case of remote object, the request is forwarded to Distributed OID Request Dispatcher. All objects not found in the local databases are considered to be remote.

Committing Transaction is only a more prioritized phase in the Transaction process execution. The Transaction Process is currently committing the transaction it is executing. The priority of Committing Transaction is higher than the priority of any other Transaction Process still in transaction execution phase. If the transaction does not conflict with other transactions, the Transaction Process writes modified data to the database and the transaction is then finally committed. Transaction commit is done, when all data modifications and index modifications are successfully stored into safe storage via the Log Writer process.

Index Updater takes care of attribute index updating during the transaction commit phase. Indexes are updated after the committing transaction is successfully validated.

Cold Data Buffer Manager receives cold data read and write requests from the OID Request Dispatchers. Requests are based on physical address of the accessed data. Read requests are first resolved from the buffer pool. When specified data item does not exist in the buffer pool, the item is fetched from the disk. Write request cause written data items to be pinned into memory. In the case of transaction commit, the modified data items are written into disk and unpinned before the commit is accepted. Thus, the disk database never contains any uncommitted data and always contains all committed data.

Distributed OID Request Dispatcher is a special database process that maintains the connection to the Distributed Database Subsystem. It receives OID read requests from local OID Request Dispatcher and forwards these requests to remote database via Distributed Database Subsystem. It also accepts incoming remote OID read requests and satisfies these requests with a co–operation of local OID Request Dispatcher.

Log Writer handles log write commands. When the Primary Node is acting normally, the requests are passed to the Mirror Node. When the Primary Node is acting as the Transient Node, log write requests are

written directly to disk. In both cases, the write process is synchronous, thus a log write operation is finished only when it is guaranteed, that entry to be written is permanently stored either on the Mirror Node or on the disk.

Hot Data Flusher writes hot data contents into disk storage, thus creating a disk copy of the main memory database. This process is normally used only in the Mirror Node, but it can be used also in the Transient Node.

5 Fault Tolerance Features in Rodain Database

The term fault tolerance has been defined in numerous ways, see for example [1] and [7]. Basically every definition carry the idea of maintaining the system's functionality inspite of some failures. This can be achieved by restoring the system to some previous point or by adding redundancy to the system. The three different types of redundancy (physical resource, time and information redundancy) all provide a different way of adding redundancy to the system.

Replication of physical resources is the most common way of adding redundancy. It is also the base of fault tolerance in the Rodain Database. We use only two similar nodes. The architecture does not hinder usage of multiple mirror nodes if required.

Time redundancy mechanism is not used in the Rodain Database. It is not very feasible solution for a real–time system where the amount of time needed for execution is important. The re–execution needs extra time, which can more easily lead to missing the deadline specially in failure situation. This can be avoided by adding extra capacity that is used only in failure situation. Since the failures are quite rare it is not very cost effective.

Information redundancy is also not used in the current Rodain Database prototype. Instead we simply assume that the information does not change due failures such as memory corruption or erroneous database update. If this assumption does not hold, then the Rodain Database needs some mechanism to maintain the information consistency.

5.1 Failure semantics of the server

The fault–tolerance of any system is based on the knowledge or assumption of the failure behaviors of the services or servers it uses. Cristian [1] calls this behavior the server's failure semantics. The servers form abstraction hierarchies, where each server masks, or tries to mask, the failures of servers it uses.

Our current implementation of the Rodain Database prototype assumes that the computing hardware has crash failures only. This means that we assume the hardware to perform correctly until it simply stops functioning. We do not allow omission or performance failures on the communication channel either. Instead we require that the communication channel also has only crash failure semantics. This can be achieved, since our prototype has multiple networks between the nodes.

We have lowered the probability of total system crash by using two separate, identical nodes. Only if both nodes fail at the same time the whole database system will stop functioning from the clients point of view. For the client application the whole Rodain Database Node may seem to have omission failures. They can occur when the Primary Node fails and active transactions are lost. The client application gets no reply and can safely assume that it never will receive any reply.

Even the failure of disk subsystem can be masked if both Primary and Mirror Node remain functional until the database copy on disk has been rebuild. Because this is expensive, takes a long time, and fault tolerant disk systems such as RAID exists commercially, we have assumed the disk subsystem used in the Rodain Database Prototype to be failure free.

5.2 Node Replication

The group of replicated server nodes can be organized in different ways. In the Rodain Database we have chosen to make the Primary and Mirror Nodes loosely synchronized. The tight synchronization is not cost effective with the failure assumptions we have made. Generally it would only add extra overhead without gaining much. The only advantage would be that the Mirror Node could continue execution of active transactions. Because the telecommunication area does not need ultra reliable real–time systems, we can use the computing power to productive work and allow the client application to see more failures.

In the Rodain Database Prototype the Primary Node executes all the operations. Mirror Node only follows its state changes, but makes the changes bit later in its own state. This reduces the time needed between failure of Primary Node and execution start on Mirror Node. By keeping the state of Mirror Node quite close to the state of the Primary Node we can meet the requirement of downtime restricted to few seconds per each failure. Other alternatives are not as appealing. If we keep the Mirror Node totally passive and have to rebuild the database contents on the Mirror Node from scratch each time Primary Node fails the downtime would be minutes instead of seconds. A more expensive solution would be to keep the database in a reliable memory such as Flash or battery backup memory, that is not lost in most hardware failures. Even if the nonvolatile memory maintains the database content, the processor is not available until the computing node has recovered. Also the usage of large amounts of nonvolatile memory requires a special computer. As the current telecommunication databases are based on special hardware we are more interested in seeing how they can be implemented on ordinary computers.

To maintain the synchronization between Primary and Mirror Nodes the state changes must be past from Primary to Mirror Node. In the Rodain Database we use transaction logs to pass information of database modifications from Primary to Mirror. Logs are the basic mechanism for databases to keep track of database modifications. Traditionally they are used to return the database back to a consistent state if something fails. Therefore in the Rodain Database the Mirror Node is continuously recovering its database copy based on the arriving logs.

Even though the main reason for passing the logs to the Mirror Node is to update the database copy on it, the Mirror Node can help the Primary Node in log handling. The Mirror Node can behave as a separate log storing processor (see [8] or [9]). It is responsible for storing the logs in a stable storage on disk. The logs on disk are needed only when either node has failed and is recovering.

The log storing mechanism on the Rodain Database differs from the mechanisms presented in [8] and [9]. They both order the logs on a page basis because they use a disk based database. Since the critical part of the Rodain Database is a main memory database we can use transaction based log ordering presented in [5]. The logs of one transaction are grouped

together and stored adjacently. This makes the recovery of failed node easier because it can follow the storage order of the logs when updating the recovering database.

6 Conclusion

In this paper we have given an overview of our prototype database system Rodain. It is designed to fulfill the requirements for Service Data Function and Service Data Point presented in the Intelligent Network concept. It is a real–time database that provides deadlines and criticality value for each transaction. It is also an object–oriented database, that allows storage of data and the methods accessing them. The time critical part of the database is stored fully in the main memory to keep the data access time short and to meet the execution deadlines of the real–time transactions.

The Rodain Database is also a fault tolerant service. It contains two nodes that can behave on either role as Primary or Mirror Node. The database service assumes that the underlying hardware and software components are either failure free or have only crash failure semantics. The fault tolerance functionality of the Rodain Database is based on the transaction logs. The logs are used to maintain the state of the mirrored database copy close to the state of the primary database. Also the logs are used when a failed node is recovering back to operation.

Acknowledgements

This work has been carried out in the research project RODAIN (1996) funded by the Finnish Technology Development Center (TEKES) together with Nokia Telecommunications, Solid Information Technology and Sonera. The author wants to thank Kimmo Raatikainen, Jukka Kiviniemi, Jan Lindström, Pasi Porkka, and Juha Taina from the Department of Computer Science in the University of Helsinki for the fruitful discussions and valuable comments during the research. The industrial partners (Jukka Aakkula and Asko Suorsa from Nokia Telecommunications, Jussi Ollikainen and Jari Vänttinen from Sonera, and Kyösti Laiho from Solid) have provided useful information and feedback comments during the project.

References

[1] *Flavin Cristian* Understanding fault–tolerant distributed systems. *Communications of the ACM*, 34(2):57–78, February 1991.

[2] *H. Gomaa* A software design method for real–time systems. *Communications of the ACM*, 27(9):938–949, September 1984.

[3] *ITU Q–Series Intelligent Network Recommendation Overview. Recommendation Q.1200.* ITU, International Telecommunications Union, Geneva, Switzerland, 1993.

[4] *ITU Intelligent Network Distributed Functional Plan Architecture. Recommendation Q.1204.* ITU, International Telecommunications Union, Geneva, Switzerland, 1994.

[5] *H. V. Jagadish, A. Silberschatz, and S. Sudarshan* Recovering from main–memory lapses. In *Proceedings of the 19th VLDB Conference*, pages 391–404, 1993.

[6] *J. Kiviniemi and K. Raatikainen* Object oriented data model for telecommunications. Report C-1996-75, University of Helsinki, Dept. of Computer Science, Helsinki, Finland, October 1996.

[7] *H. Kopetz and P. Veríssimo* Real time and dependability concepts. In S. Mullender, editor, *Distributed Systems, second edition*, pages 411–446. Addison–Wesley, USA, 1993.

[8] *T. J. Lehman and M. J. Carey.* A recovery algorithm for a high–performance memory–resident database system. In U. Dayal and I. Trager, editors, *Proceedings of ACM SIGMOD 1987 Annual Conference*, pages 104–117. ACM SIGMOD, ACM Press, May 1987.

[9] *E. Levy and A. Silberschatz* Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, December 1992.

[10] *K. Raatikainen and J. Taina* Design issues in database systems for telecommunication services. Report C-1995-16, University of Helsinki, Dept. of Computer Science, Helsinki, Finland, September 1995.

[11] *K. Ramamritham* Real–time databases. *Distributed and Parallel Databases*, 1:199–226, 1993.

[12] *J. Taina and K. Raatikainen* Experimental real–time object–oriented database architecture for intelligent networks. *Engineering Intelligent Systems*, 4(3):57–63, September 1996.

[13] *P. S. Yu, K.-L. Wu, K.-J. Lin, and S. H. Son.* On real–time databases: Concurrency control and scheduling. *Proceedings of the IEEE*, 82(1):140–157, January 1994.