University of Helsinki
Department of Computer Science
Series of Publications C, No. C-1995-42

On Implementing a Prototype IN Database Architecture*

Juha Taina and Harri Töhönen

Helsinki, November 1995

Report C-1995-42

University of Helsinki
Department of Computer Science
P. O. Box 26 (Teollisuuskatu 23)
FIN-00014 University of Helsinki, FINLAND

* This report is based on work done in Darfin-project (Project No 30018) funded by Telecom Finland.

On Implementing a Prototype IN Database Architecture†

Juha Taina and Harri Töhönen

Report C-1995-42
nn pages
Helsinki, November 1995

Abstract. Databases in telecommunications have gained growing attention in research field. New telecommunications services need efficient and flexible database architectures. We have designed a database architecture that is especially suitable for Intelligent Network use, but that can also be used on other telecommunications areas. Our architecture is a distributed real-time object oriented database architecture where each distributed database node is able to run alone. From the architecture we designed and implemented a simple prototype database management system. In the prototype we concentrated on single node database operations.

Key Words: keyword

CR Classification: C.2.1 H.2.4

†This report is based on work done for Darfin-project (Project No 30018) funded by Telecom Finland.

Contents

1. Introduction

In the recent years databases in telecommunications have gained a growing attention in research field. One of the major areas is Intelligent Network (IN) architecture and database use in IN services. In project Darfin (Database ARchitecture For Intelligent Networks) we wanted to design and implement a suitable database architecture for IN use. This report explains our final result: the first version of our prototype database implementation for IN.

In [Raat94] and further in [Tain95] we have identified the needs that the ITU-T recommendations list for IN database services and designed the architecture accordingly. The most important requirements are object orientation, real time access to data, and fault tolerance. Starting from the requirements we designed the first version of our database architecture. The architecture is especially suitable for IN use but its major ideas are also useful for a Telecommunications Management Network (TMN) database architecture. The designed architecture is a distributed real-time object oriented database architecture that may be used for implementing a telecommunications element that needs database services, such as a service data function (SDF). It is not a complete network element architecture. Instead it can be used to implement embedded database services in a telecommunications element. It can also be used as a stand-alone architecture that serves several applications.

The final architecture is still under evolution. We noticed that there are several areas where we need to have more information for the architecture. We wanted to learn more of combining real time transactions and object orientation, and we wanted to learn implementation methods for SDFs and other network elements using our database architecture. Due to this we implemented a simple prototype database architecture that follows the general ideas of our architecture and yet is simple enough for fast implementation. This report describes our work on the prototype database.

The rest of the report is organized as follows. In Section 2 we describe the logical architecture of our database. The section describes the background of our database implementation, it is not a full description of the architecture. In Section 3 we describe the general architecture of our implementation. The general level describes a single node architecture of a full distributed database management system. In Section 4 we describe the interpreter subsystem that is used for interpreteting various user requests to an uniform database language. In Section 5 we describe the actual database management subsystem that is responsible of physical data and transaction management. In Section 6 we describe our test application and the designed interface to the database node. In Section 7 we will discuss about experiences that we got from the implementation. Section 8 is the summary.

2. Darfin logical database architecture

The background work for our logical database architecture is described in [RaTa95]. It includes principles of real-time database systems, areas of concurrency control, main memory databases overview, and correctness criteria of transactions, all of which play an important role in our architecture.

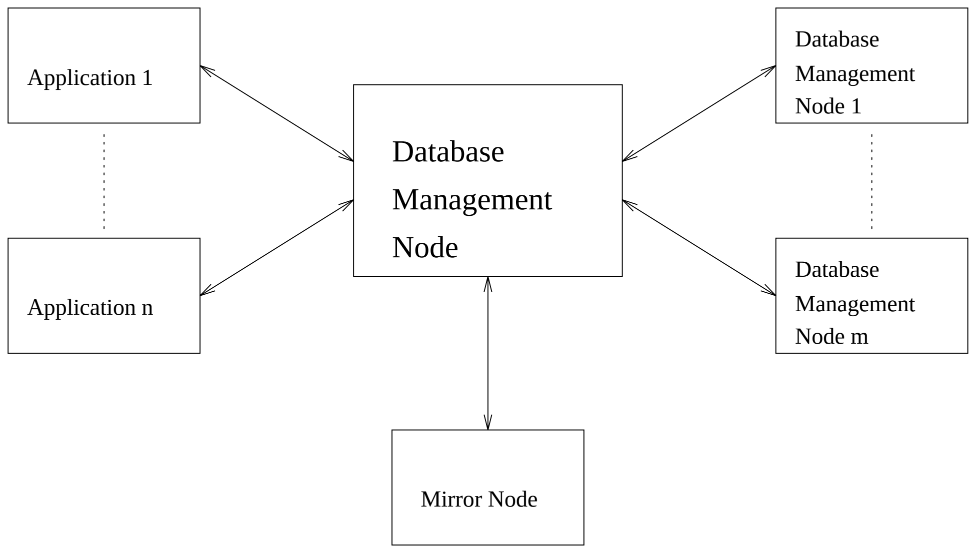

Figure 1: Overview of the darfin database architecture

The logical model of our database architecture is a distributed object-oriented real-time database architecture where every database node is replicated into a mirror node. Every node may communicate with several applications and with several other database nodes (figure 1, from [TaRR95]). The architecture is described in detail in [TaRR95]. The logical model definition also defines three major areas in the architecture: distribution model, logical object-oriented data model, and transaction model.

Our distribution model is based on two assumptions. First, we believe that most operations are local. Second, when a distributed operation occurs it is usually relatively simple and concerns only few objects in the system. These assumptions simplify distribution modeling because it allows us to concentrate on database nodes like they were primarily stand alone databases and secondary nodes in a distributed database. The other alternative, using complex distributed operations, would have meant lots of traffic between database nodes and difficult transaction control between several nodes. Unfortunately we do not know for certain if our assumptions are right.

The logical object-oriented data model is based on Object Database Management Group standard ODMG-93 [Catt94]. It is a recommendation for an object-oriented database interface standard. The ODMG-93 recommendation is the first effort to standardize object oriented database interfaces. It is based on the object model of Object Management Group (OMG) which itself is becoming a globally accepted standard for distributed object processing. In [Tain94] we compared OMG, ODMG, X.500, and X.700 object models, and in [TKMR95] we expanded the comparison work to IN environment. Our conclusion was that ODMG-93 object model is strong enough for our use. We could have implemented a private object model but then we would have lost compatibility with future commercial products.

The transaction model is based on real-time transactions. A transaction may or may not have a deadline that must be met. There are four types of transactions.

1) *Hard transactions* that must always meet the deadline. A system that uses hard transactions

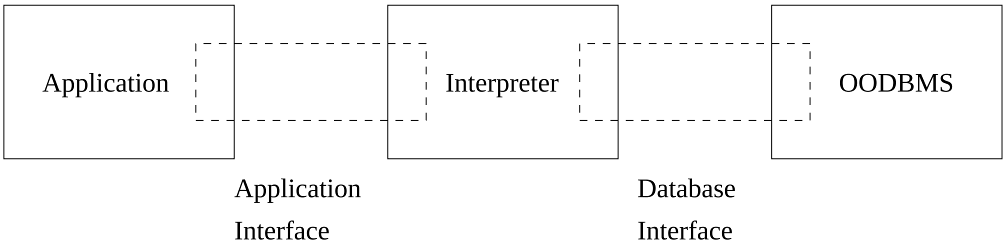

Figure 2: Communication channels

must prove that the transactions will always meet their deadlines. We do not support hard transactions in our architecture.

2) *Firm transactions* that are killed when the deadline occurs. Most of our deadlined transactions are of this type.
3) *Soft transactions* that are finished on a lower priority after the deadline occurs. Currently we are not using this type of transactions.
4) *Transactions without deadlines* that behave like regular transactions or like soft transactions after deadline. Some of our transactions are of this type.

Every transaction consists of a list of subtransactions that are executed in a serial order. A subtransaction may or may not have a deadline. For clarity reasons we call subtransaction deadlines timeouts.

A subtransaction may belong to any of the supported transaction types. If a subtransaction timeout occurs, the subtransaction is killed. The whole transaction, however, is not killed until the transaction deadline is over. The transaction model allows several subtransactions in a transaction but only one of them is active at a given time. Every subtransaction returns a result, either a value from the database or an operation result status such as an acknowledgement.

The architecture consists of several cooperating database subsystems that can be replaced as long as the protocol between the subsystems remains unaltered. A full description of the subsystems is in [TaRR95]. The prototype uses only two subsystems: an interpretation subsystem and a database management subsystem.

The interpreter subsystem is responsible of interpreting application requests to the database subsystem. Using an interpreter allows the architecture to use various user interfaces without changing the actual database implementation.

The interpreter is responsible of two interfaces (figure 2). First, it maintains an application-specific view to the database for the applications that use it. As it is possible to use different interpreters, it is also possible to use different application views. Second, it maintains a globally uniform view from and to the database management system. The database management system itself uses only a single protocol and database interface to the interpreters. The interpreter is responsible for translating application requests to the database management subsystem and database responses to the application.

The database management subsystem is responsible of physical data and transaction management. It is the core subsystem of a database node.

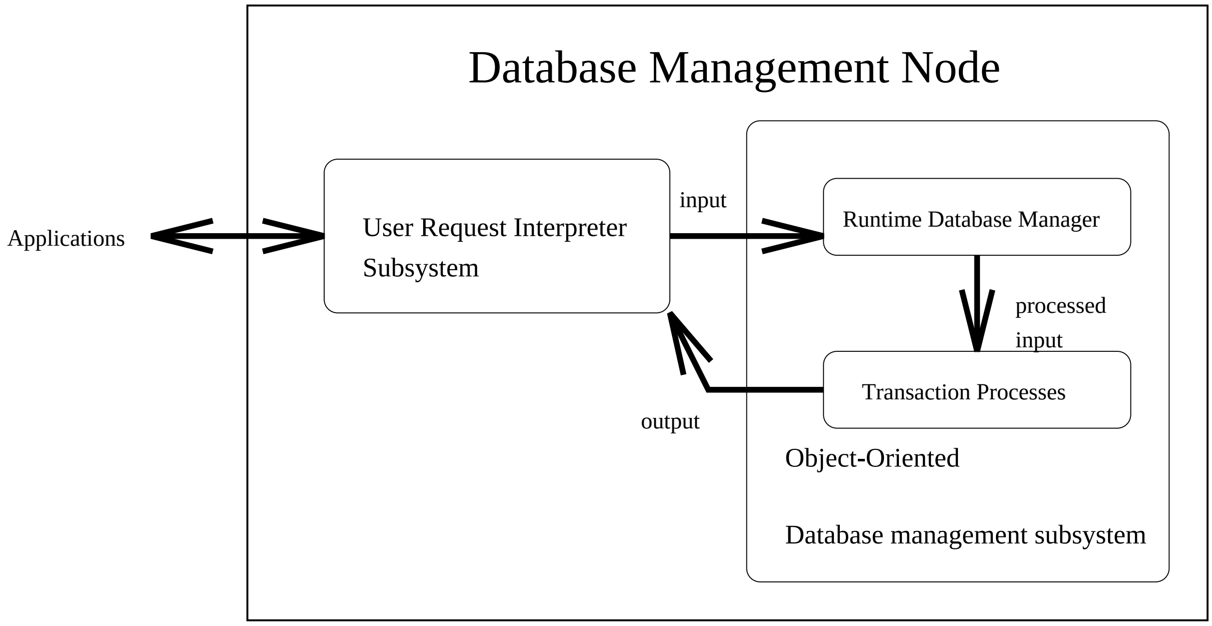

Figure 3: Prototype database node

3. General prototype structure

The prototype implementation simplifies the general architecture in distribution and fault tolerance. The prototype is a single node database that can communicate with several applications. It does not support distribution, mirror nodes and fault tolerance.

Distribution and distributed operations are possible to recognize in the prototype. In the future versions of the prototype we might add full support for distribution control.

Fault tolerance is not supported in the current version of the prototype. Our fault tolerance research is on early stages. Adding fault tolerance support to the prototype would have cost too much compared to the additional gained information.

In the prototype implementation we concentrated on a single node architecture. We implemented the node architecture in a Linux platform. The platform follows POSIX standards with additional BSD UNIX support for network streams and process to process communication. The different subsystems are UNIX-processes.

In a real-time database management system the predictability of process execution time is of major importance. Also it is important to predict data access times. Unfortunately a POSIX-compliant system cannot answer either of the important areas. However, we wanted to learn about real-time database implementations without learning about new operating systems, or worse yet, implementing a complete new operating systems. Thus we used a Linux platform with its drawbacks.

In the prototype we implemented two subsystems: interpreter subsystem and a database management subsystem (figure 3).

4. Interpreter subsystem

The interpreter subsystem interprets application requests to the database management process. The implementation uses UNIX processes by creating a new interpreter subsystem process for each application connection. The chosen architecture decision simplified the interpreter implementation, but it also caused unnecessary UNIX processes to the system. The less UNIX processes the system has, the better predictability the database management subsystem has. In the future the interpreter should not create more than a single process.

Applications communicate with a database node via BSD network sockets. The interpreter process is the only process that communicates with the applications. It receives requests, interpreters them, and sends them to the database management process. The communication between the interpreter process and the database management process is via unix sockets.

The interpreter subsystem follows the logical interpreter model described earlier. It accepts application interface commands and interprets them into database interface commands.

Our interpreter implementation was mostly straightforward. Due to resource limits and lack of time we first designed a suitable application protocol and then used it almost directly in the database interface. The resulting protocol goes as follows.

- A new application initiates communication with the interpreter process. The interpreter process allocates a new communication channel for the application and passes acknowledgement.
- The application sends an open database command to the interpreter. The interpreter forwards the command to the database management subsystem. An acknowledgement is then returned to the application.
- The principal item of the communication is a transaction. Every database operation from the application is embedded into a transaction. The model follows the logical transaction model that we described earlier.
- Every embedded database operation has its own timeout. The subtransaction may rollback if it cannot meet its timeout. The whole transaction may rollback if it cannot meet its global deadline.
- The interpreter interprets the subtransaction request and then sends it to the database management system. The database management system processes the subtransaction and sends a result to the interpreter. The interpreter interprets the result and sends it to the application. Only after this the application may request another subtransaction.
- When the application wants to finish the current transaction, it either sends a commit request or a rollback request. The commit request commits the changes to the database, the rollback request discards them.
- The application process may only send a single transaction or a subtransaction at a time. After commit or rollback request the application may start a new transaction.
- The last command from the application is a close request. It closes the initiated connection. The database is no longer available to the application.

The most complex section of the interpreter is interpretation from an Object Query Language (OQL, see [Catt94]) query to the database application language. OQL is based on expressions and operations between them. Each operation creates a new expression that is either a value or an object. An object may also be a container object that consists of several objects. The prototype implementation uses only values and atomic objects. The implementation builds an expression tree of a query and sends the tree to the database management subsystem.

The OQL interpreter is implemented using unix tools flex and yacc. Although the interpretation algorithm was implemented and tested, the actual interface to the database was not implemented.

5. Database Management System

The database management subsystem is the major component of the prototype. It is responsible of maintaining the object oriented database and real-time requirements.

The general database management system architecture may be seen in figure 4. The architecture consists of a set of database processes that use database management system services. The services are divided into four layers.

The database management system is based on processes that the system schedules. The prototype has three kinds of subprocesses: runtime manager process, real-time scheduler, and transaction processes.

- *Runtime Manager process* is responsible of processing new connection requests, creating transaction processes, and forwarding transactions to transaction processes. When a new request arrives, the runtime database manager resolves the receiver transaction process and awakes it. The transaction process is then ready until the real-time scheduler grants it access to the database management services.
- *Real-Time Scheduler* is responsible of scheduling transaction processes. Only a single transaction may be active at a time so there is no real concurrency control. However, when a transaction with a higher priority occurs the running subtransaction is rollbacked.
- *Transaction processes* execute user transactions from the interpreter. A transaction process receives input from the runtime manager and sends output directly to the interpreter subsystem. The interpreter then interprets output to the actual application.

The services that the database management system offers to its processes may be divided into four layers (figure 4).

- *Database interface layer* consists of services that start transactions and support processes. The transaction manager starts new transactions and subtransactions, the process manager offers services for process control, and the command interface processes subtransaction parameters and sends the processed commands to the command interface layer.
- *Command interface layer* processes the received commands and forwards them to the physical object manager. It interprets the processed commands into several function calls that are forwarded to the object layer.
- *Object layer.* The object layer implements physical objects and operations to the objects. The object model used follows roughly ODMG-93 standard [Catt94] except that it is simplified in the following areas.
  - The objects may have only a single name. ODMG-93 allows objects to have several names
  - Only atomic attributes are supported. ODMG-93 allows structures and container classes.
  - No inheritance.

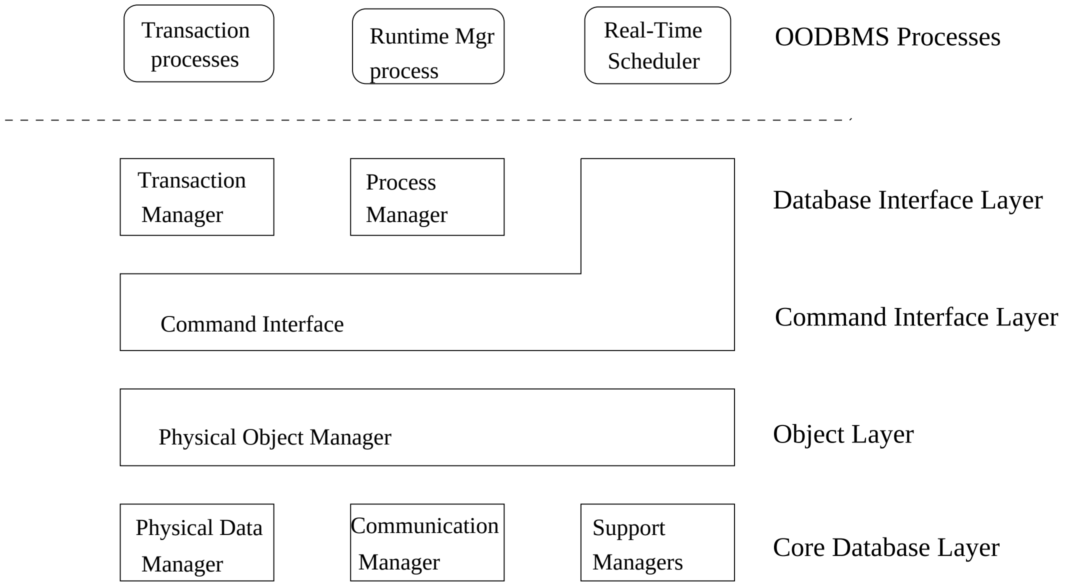

Figure 4: Database management prototype – processes and layers

  - No object operations

  The model supports object names, object identities, and automatic name and identity object directories.

- *Core database layer* implements core operations to the database management system. It consists of a physical data manager, a communication manager, and a support manager.

  The physical data manager is responsible of maintaining physical data. The supported model is a pure main-memory database. We use secondary storage only when loading or storing the main memory database image to a disk.

  The communication manager maintains communications between the database management system and the interpreter process.

  The support manager consists of miscellaneous functions that may be used on the other managers and processes. It contains operations for list handling, memory allocation and log management.

The control processes, runtime manager and real-time scheduler, control transaction processes that execute user commands to the database. The process control goes as follows (figure 5).

1) Both the runtime manager process and the real-time scheduler are signal handlers. The former is activated when a new transaction or subtransaction request arrives from the interpreter, the latter is activated ten times a second.
2) When new request arrives, the runtime manager process reads the request from the communication channel and awakes the appropriate transaction process. If the communication channel request is new the runtime manager process creates a new transaction process
3) A transaction may or may not have a deadline. If it has a deadline, the deadline defines its priority. We use the earliest deadline first -method where the earliest deadline has the highest priority. A transaction without a deadline has the lowest possible priority.

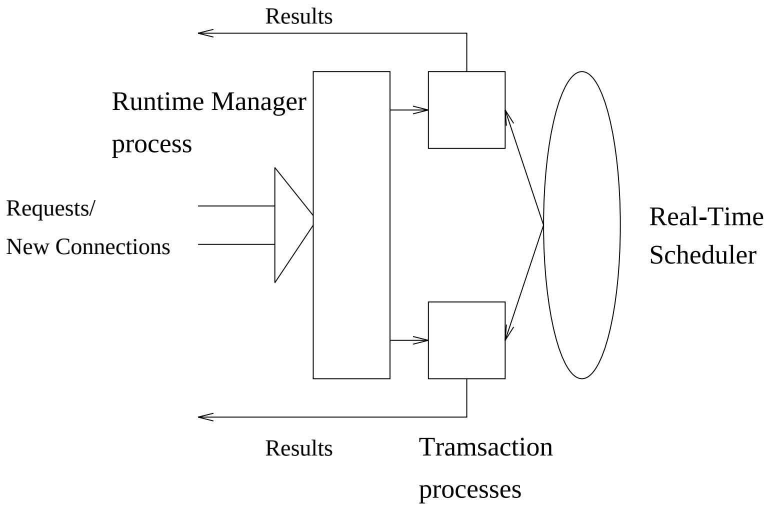

Figure 5: Process control in database management prototype

4) The real-time scheduler keeps track of transaction deadlines and priorities. The transactions are executed in a serial order. Thus no concurrency control is used. If a new higher priority transaction arrives the current running subtransaction is rollbacked. Also if the real-time scheduler notices that a deadline has been reached it rollbacks the executed transaction. If a subtransaction timeout has occurred, the real-time scheduler rollbacks the current subtransaction.
5) The transaction processes receive preprocessed data from the runtime manager process. The transaction processes never read the communication streams. However, they write the results of transactions and subtransactions to the channel.

6. Application

In the prototype design we also designed and implemented a test application that may be used with the database architecture. The designed test application is a CMIS agent that understands XI protocol. XI is an implementation of CMIP protocol and X.700 information model that is used in Telecom Finland.

The test application consist of the following items.

- *The used user interface* is a character based XI message sender and receiver.
- *The XI Generic Layer* consists of the XI interface. It understands a large subset of ITU-T object management X.700 messages.
- *The XI Server specific Layer* converts the used application model to the real network resource. In the test application it also converts XI operations to object database operations.
- *The Database Interface* hides the communication channel between the application and the database architecture. It allows communication over the network. The application and the database management system do not have to be in a same network node.

The lowest layer of the application implements an interface to the database application request interpreter. The actual implementation is a C++ class library that both the application and the interpreter use for mutual communications. The class library consists of a transport section that implements communications via network sockets, and a message section that implements different message types, objects, and coding functions.

The used protocol follows low level ODMG-93 standard operations [Catt94]. The messages belong to the following classes.

1) *Database communication management* that consists of operations for opening and closing a database.
2) *Transaction management* that consists of operations for starting, committing and rollbacking transactions. The test application may use real time constraints in the database transactions.
3) *Object operations class* that consists of operations for finding object identities from an object name, finding objects when object identity is known, and managing object names (setting names, getting names and renaming).
4) *Object management operations class* that consists of operations for creating, deleting and updating objects.
5) *Object Query Language management* that consists of operations that allows the application to send ODMG-93 Object Query Language queries to the database. Within its operations the test application may create dynamic OQL queries.

7. Implementation experiences

Due to short implementation time we could not concentrate on all interesting areas. The major interest we had was of the general implementation experiences and we concentrated on it in the prototype. This proved to be a good approach.

The main results we got are the following.

- *A Pure UNIX-based platform is insufficient for a real-time database management system implementation.* At least the following aspects cause problems in UNIX.
    1) *Process control.* A real-time database management system needs total process control of the system. A pure UNIX-based system, however, does not allow an application to have process control. An approximate simulation might have been possible if we had used different nice-values that affect the unix process priorities and used shared memory between processes. We did not want to go to this unnecessary detail as it would have been a poor substitute for true real-time process control. Even in a prototype level the database management system must be able to create its own processes and schedule them accordingly. Something close might have been possible with Solaris threads, but in the Linux system we did not have them.
    2) *Memory management.* In our prototype we created a main memory database management system that uses UNIX virtual memory services. In small databases this kind of approach works well. However, in a larger main memory databases and in real-time databases where we have to give estimates of data access times this kind of approach is not good enough. Also according to [KLVA93] virtual memory often works poorly on database management systems.

3) *Secondary storage management.* Like main memory control, the secondary storage control (usually disks) must be processed locally in the database management system. We did not use secondary storage in our prototype. Rather, all information was in main memory.
4) *Operating system calls.* It is difficult to estimate operating system call lengths even in simple cases. Still we need the information if we want to use any other scheduling algorithm than the simplest first come first served and earliest deadline first.

- *It is better to build a real-time object oriented database than an object-oriented real-time database.* The approach direction to the dilemma turned out to be very important. When we started from an object-oriented database approach and tried to add real-time aspects to the architecture we turned into certain problems. First, a full object-oriented paradigm is complex to implement and needs a lot of work even in a simple prototype. Such problems as adding object inheritance, object operations, and late binding are complex tasks to do well. Adding real-time requirements to such an architecture leads into new scientific problems. For instance, what is the level where real-time requirements should be seen? Is it on transaction level, or perhaps on an object level? What to do with inheritance? The problems and open questions are various.
  We got better results when we started from a real-time database approach. A real-time database model in its simplest form is rather simple and easier to implement than an object-oriented database model. When we had a working earliest deadline first -real-time database management system, we added enough object oriented areas into the architecture to call it a simple object-oriented database. The prototype is not a full object-oriented database management system, it is not even a full real-time database management system. But it is a lot closer to a real-time database management system than an object-oriented database management system. Furthermore, it is closer to the requirements that were designed during project darfin than if we had started from object orientation. There are already several object-oriented databases publicly available. Their object models are sufficient for IN use. Yet there are no real-time databases that fit IN use in market today. The best products are tailored for a specific tasks and are often used in closed laboratory environments.
- *Even a simple prototype database management system needs a lot of work and resources.* The prototype architecture that we implemented was written in two months, except some parts of the test application that were reused. The result was about 15,000 lines c-code, and yet we implemented only a bare core of a full real-time database management system. In the future versions of the prototype, giving enough project design and implementation time is a necessity for success.

8. Summary

In project darfin we have worked on a database architecture for Intelligent Networks. The final phase of the project was a simple real-time database management system prototype for Intelligent Network usage. The prototype is a single node equivalent of a larger distributed database architecture. It is implemented in a POSIX-compliant Linux system with extra BSD network interface. The implementation follows our designed database management system architecture with simplifications that necessary for finishing the project in time.

Our main result of the implementation project is that even a simple prototype database that is based on real-time requirements and object orientation needs a special real-time implementation platform, and extremely lot of work.

References

[Catt94] Cattel R. G. G. (Ed), Object Database Standard: ODMG–93 Release 1.1. Morgan Kaufman Publishers, 1994

[KLVA93] Krueger K., Loftesness D., Vahdat A. and Anderson T., Tools for the Development of Application-Specific Virtual Memory Management. ACM SIGPLAN Notices 28,10 (1993), pp. 48 – 64

[Raat94] Raatikainen K. E. E., Information Aspects of Services and Service Features in Intelligent Network Capability Set 1. Technical report, University of Helsinki, Department of Computer Science, 1994

[RaTa95] Raatikainen K. E. E. and Taina J., Design Issues in Database Systems for Telecommunication Services. Proceedings of IFIP IN'95 Conference. IFIP, 1995

[Tain94] Taina J., Evaluation of OMG, ODMG, X.500, and X.700 Data Models. Technical report, University of Helsinki, Department of Computer Science, 1994

[Tain95] Taina J., Requirements Analysis for Database Services in Telecommunications. Technical report, University of Helsinki, Department of Computer Science, 1995

[TaRR95] Taina J., Rautila M. and Raatikainen K. E. E., An Experimental Database Architecture for Intelligent Networks. Proceedings of IFIP IN'95 Conference. IFIP, 1995

[TKMR95] Taina J., Karttunen T., Martikainen O. and Raatikainen K., Evaluation of Database Architectures for Intelligent Networks. Technical report, University of Helsinki, Department of Computer Science, 1995