Proceedings of the real-time database workshop, Eindhoven, 23 February 1995

Eindhoven University of Technology
Department of Mathematics and Computing Science

Proceedings of the Real-Time Database Workshop [1]

by

editors: P.D.V. van der Stok and J. van der Wal

23 February 1995

[1] supported by STW in connection with project EIF33.3129

ISSN 0926-4515

Computing Science Report 95/27
Eindhoven, August 1995

Proceedings of the Real-Time Database Workshop [1]

editors: P.D.V. van der Stok J. van der Wal

23 february 1995

[1]supported by STW in connection with project EIF33.3129

Summary

Introduction

The purpose of this workshop was to introduce the members (3 companies and 2 research institutes) of the user committee and the team members (3 staff and 2 PhD students) of the STW-project: *Construction and performance of real-time transactions* to each other. The user group represents a large group of (potential) users of real-time databases. During the workshop insight was gained in the requirements of the companies they represent. Four talks were presented by different companies to state their requirements on on-line databases. A talk was given by INRIA about maximum response times of transactional systems. A representative of STW was present to explain the tasks and procedures relevant for the members of the User Committee as laid down by the STW organization.

The project is motivated by two observations: (1) more and more response times are specified for database accesses and (2) recently, real-time applications demand so much structured data that database techniques are needed to manage these data.

The specification of deadlines is necessary for real-time systems because often the action on the process under control must be carried out before the process under control enters a certain state. A well known example is the activation of a manufacturing machine at the moment the manufacturing object passes on the conveyor belt.

Database applications start to necessitate deadlines to satisfy customer demands. The deadlines give guarantees on the performance of the on-line database accesses. When the deadlines are not met, customer satisfaction may drop such that customers turn to other installations where these deadlines are met. Considerable financial loss may be suffered when the deadlines are not met. An example is: large quantities of money are transferred with a slight delay.

For the purpose of this project the term real-time transactions is used to indicate those transactions which need to meet their deadlines. A real-time database is a database where real-time transactions are executed. The performance of the database is no longer determined by the mean duration of the individual transactions but by the number of transactions which meet their deadline. Figure 1 shows that these two measures lead to different results. Two density functions are represented. The first one has the lower mean but a higher variance and a larger fraction of transactions that do not meet their deadline than the second one. The mean of the elongated distribution is smaller but the larger number of transactions which meet

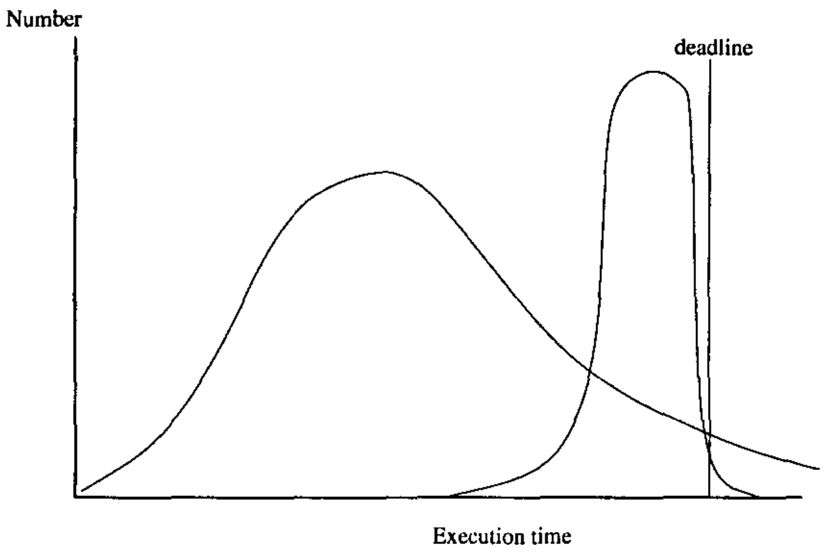


Figure 1: distribution of hypothetical transactions

their deadline is created by the narrow distribution. This figure indicates that different techniques from the current ones are probably needed to satisfy the real-time requirements of the database. During this project we will try to calculate the distribution of the transaction execution times and develop appropriate algorithms to increase the real-time performance.

In these proceedings, papers supporting the presented talks and copies of the presented overhead sheets are bundled. Their contents represent the interests of the users in the project and the current state of analysis done by project members.

A first talk was given by L.J.M Nieuwenhuis of KPN Research, titled: *Database Requirements in Telecommunications Systems.*

In this tlak the importance of databases for current and future telecommunications systems was discussed. Large amounts of information for network management, service provisioning and billing and pricing are needed. These applications put strong performance and reliability requirements on the involved database systems. Proposed implementation strategies are based on partitioning and replication. Partitioning is needed to support simultaneous, real-time transactions and replication is needed to provide reliability through fault tolerance. An overview of strategies with various levels of consistency was presented.

The second talk was presented by J. van der Meer of Ericsson Telecom, titled: *Real-Time databases: present state and future expectations.*

This talk provided a follow-up to the former talk. Given the low performance of present day databases and given the specific characteristics of the transactions requested for Intelligent Networks, Ericsson has started to build a special purpose database based on the language Erlang. The protocols, the transaction characteristics and the performance requirements of the database were presented.

The third talk was presented by A.A. Vreven of Rabofacet, titled: *Database requirements for on-line financial transactions.*

A short overview of the needs of the Rabobank are discussed. The bank is not interested in highly complex technical solutions but wants to handle on-line transactions more cheaply than its competitors. For that purpose the functional units of the database applications must be distributed over a network of teller machines and other mainframes or workstations. The Rabobank wants to know the performance consequences of these distributions.

The fourth talk was presented by P. Minet of INRIA/Rocquencourt, titled: *Maximum response time in a real-time distributed transactional system.*

Distributed transactional real-time systems pose the triple problem of serialisability, reliabiliy and timeliness. In the talk conditions necessary for the realization of these systems are determined. In this way the system constructor can verify beforehand whether the system will meet its timeliness requirements and knows the length of the transaction wait-queues.

The fifth talk was given by I. Willers of CERN, titled: *Database requirement for on-line analysis of High Energy Physics experiments.*

A short introduction to CERN was given focussing quickly onto the on-line analysis and a look at LHC where we see a growing need for computing power and large quantities of data. The first steps taken in a project which has begun with a case study of on-line event reconstruction using a parallel in-memory database were discussed in detail. A data driven approach results in the writing of scalable parallel programs and involves minimal changes to legacy code.

The sixth talk was given by Maarten Bodlaender of TUE, titled: *Real-time distributed databases.*

The talk consisted of two different parts. First the difference between normal databases and real-time databases was addressed. The field of real-time databases resulted from the joining of databases with real-time scheduling systems. These different fields have different aspects that cannot be trivially combined. Next, different ways were discussed on how these fields can be integrated to create real-time databases. It is hard to create a general purpose real-time database that satisfies all requirements that different users have. A quick look was taken on the different features that real-time databases could offer.

The seventh and last talk was presented by Simone Sassen from TUE, titled: *Performance Analysis for Real-Time Databases.*

In this talk, the uncertainty in real-time databases was discussed. In most applications of real-time databases, no exact information is available about the arrival times, the sizes and the deadlines of future transactions. Due to this uncertainty it is not sufficient to measure the performance of a real-time database in terms of the *average* response time of a transaction; one has to obtain an approximation for its *distribution.* The ultimate result should be to find an approximation for the probability that a transaction meets its deadline.

Overview of Databases Requirements for Intelligent Networks *

Willem Jonker and Lambert J.M. Nieuwenhuis
KPN, PTT Research, Groningen, The Netherlands
e-mail: { w.jonker | l.j.m.nieuwenhuis } @ research.ptt.nl

Abstract

We discuss requirements and techniques for the Service Data Point (SDP) in Intelligent Network (IN) architectures in telecommunications systems. The SDP must provide real-time, simultaneous access to high volume databases. We give a short overview of SDP requirements derived from currently developed and future IN telecommunications services.

1 Introduction

The Intelligent Network (IN) architecture [2] has been developed to improve the slow and costly process of introducing new telecommunications services in today's telecommunications systems requiring software updates in all switching systems of the public network. The basic idea of the IN architecture is to shift the service control out of the switches to a small number of Service Control Points (SCPs) somewhere located in the public network. In an IN architecture, switching systems provide a set of basic switching functions. New services can be introduced by adding a new service control program in the SCP rather than updating the software of the switching systems across the entire network.

An example of an IN service is *short number dialing* in a Virtual Private Network (VPN). The IN service request of a subscriber is detected by the switch, which then routes the request, including the 'short number', to the SCP. For each subscriber, the SCP maintains a tabel in which the 'short numbers' are mapped on the 'long numbers'. The SCP sends the 'long numbers' back to the switch, which proceeds with a normal call setup, using the received information.

Obviously, the SCP needs a repository function to store the large amounts of data needed to provide various telecommunications services to all subscribers of the public network.

The storage function of the SCP is provided by the Service Data Point (SDP), the physical entity supporting the SCP. If we assume that in future each subscriber will use at least a few IN services, an SDP has to store enormous amounts of data. In practice, an SCP processes a large number of service requests in parallel. Obviously, the performance of the SCP and SDP is directly related to the Quality of Service experienced by the service end-users. Consequently, the SDP can be characterised as a high capacity database which has to support simultaneous, real-time transactions.

*Short Note Paper for Joint Workshop on Parallel and Distributed Real-Time Systems. April 22 – 24, 1995, Santa Barbara, California

The SDP (and SCP) have to meet high reliability and availability requirements, as SDP faults may cause network wide service failure affecting all subscribers. In order to meet these requirements, the SDP implementations will be based on replication strategies to support fault tolerance.

Hence, the requirements for the SDPs in IN are challenging: highly reliable database systems must provide simultaneous real-time access to a large numbers of users. These challenges make databases for telecommunications systems an interesting subject for academic and industrial research[8, 5, 6, 9].

The objective of this paper is to give an overview of the required database functionality in IN networks (Section 2). We then extend the requirements analysis for future telecommunications services, e.g., mobile communication (Section 3). The requirements analysis justify future research on distribution, partitioning and replication techniques for databases in various applications areas of telecommunications (Section 4).

2 IN database requirements

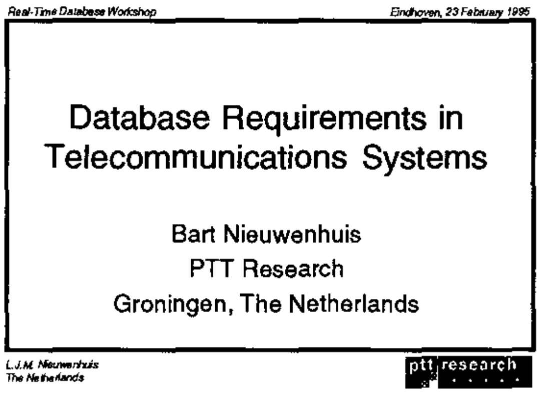
Most of the requirements presented here, are from an EURESCOM foresight study carried out in 1992[1]. Some observations are the results of an American study by Nicholas Roussopoulos[7], presented at a workshop in Germany in April 1994[5].

The requirements reported here are derived from a number of IN services under development (Freephone, Split Charge, and Virtual Private Network), and future services (mobile services, UPT, directory services) as well as a number of specific network management services (accounting and billing):

Storage requirements The data involved is rather simple and tabular representation suffices. The amount of data to be stored is expressed in storage capacity per million lines (since one SCP database per million lines seems achievable) and estimated to range from 1 to 100 Mbytes for current services.

Access requirements For current services, simple but fast retrieval is the main mode of operation. The amount of data involved is relatively small, for example for credit card calling 25 to 50 bytes per access, while for freephone numbers a service profile of about 2 Kbytes is retrieved.

Transactions requirements Most IN services will use small, real-time, read-only transactions. As far as consistency is concerned, requirements cover the whole spectrum from degree 0 (e.g., for

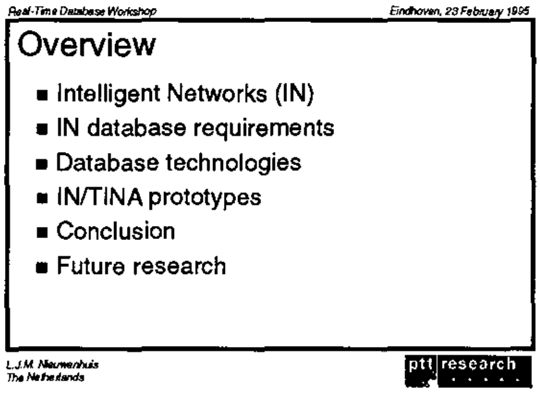


Real-Time Database Workshop Eindhoven, 23 February 1995

Database Requirements in Telecommunications Systems

Bart Nieuwenhuis
PTT Research
Groningen, The Netherlands

L.J.M. Nieuwenhuis
The Netherlands
ptt research

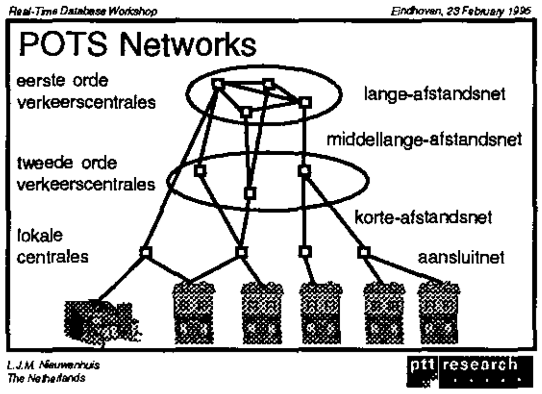


Real-Time Database Workshop Eindhoven, 23 February 1995

Overview

- Intelligent Networks (IN)
- IN database requirements
- Database technologies
- IN/TINA prototypes
- Conclusion
- Future research

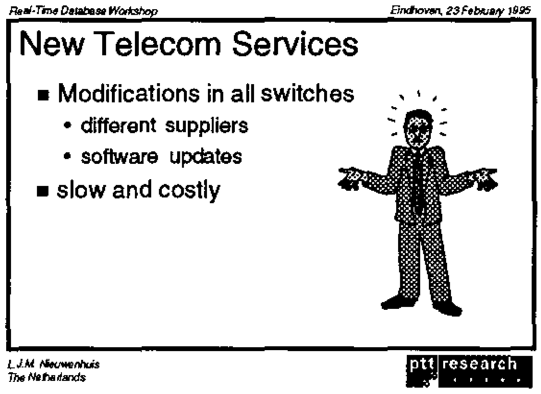
L.J.M. Nieuwenhuis
The Netherlands
ptt research



Real-Time Database Workshop Eindhoven, 23 February 1995

POTS Networks

L.J.M. Nieuwenhuis
The Netherlands
ptt research

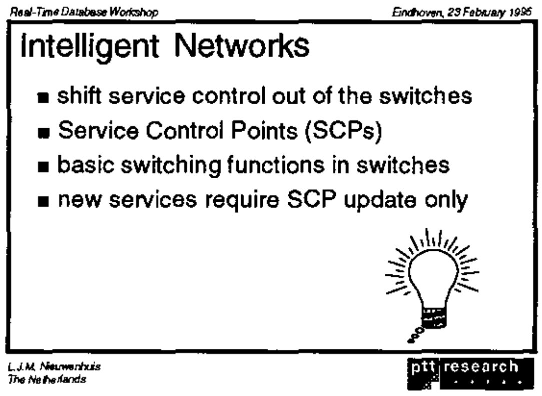


Real-Time Database Workshop Eindhoven, 23 February 1995

New Telecom Services

- Modifications in all switches
  - different suppliers
  - software updates
- slow and costly

L.J.M. Nieuwenhuis
The Netherlands
ptt research

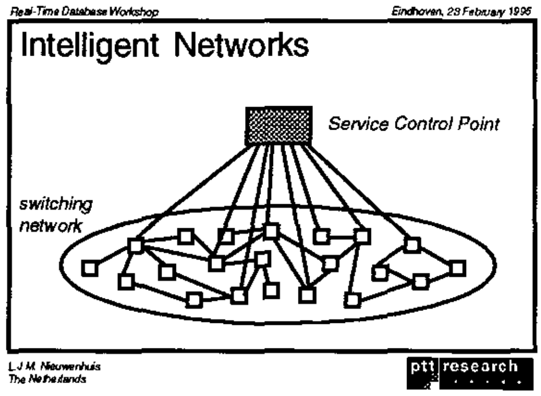


Real-Time Database Workshop Eindhoven, 23 February 1995

Intelligent Networks

- shift service control out of the switches
- Service Control Points (SCPs)
- basic switching functions in switches
- new services require SCP update only

L.J.M. Nieuwenhuis
The Netherlands
ptt research



Real-Time Database Workshop Eindhoven, 23 February 1995

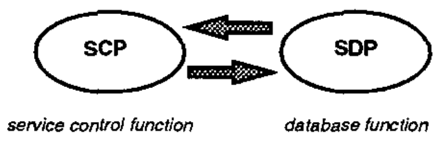
Intelligent Networks

L.J.M. Nieuwenhuis
The Netherlands
ptt research

Short Number Dialing Service

- switch detects SND request #x
- send SND(x) to SCP
- look for 'long number': y=f(x)
- send y to switch
- proceed call setup with #y

Service Data Point



Service Data Point

- multiple IN services
- millions of subscribers
- Quality of Service?
  - concurrency
  - real-time
  - reliability

IN Database Requirements (1)

- storage (1 SCP per million lines)
  - current services: 1 - 100 Mbytes
  - future services: 1 Gbyte - 1 Tbyte
- access
  - credit card calling: 25 - 50 bytes
  - freephone numbers: 2 Kbytes

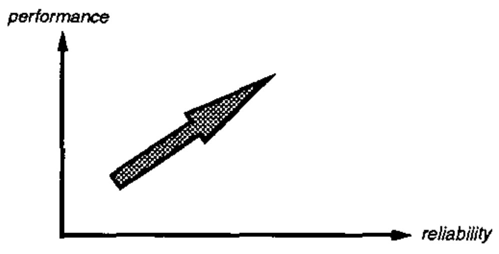
source: EURESCOM S23 project

IN Database Requirements (2)

- transactions
  - many IN services: read only
  - other applications of SDP:
    - noninterruptable, prioritised, non-wait
- performance (1 SCP per million lines)
  - current services: 20 operations per second
  - future services: 2000 -10000 writes per second

source: EURESCOM S23 project

Quality of Service



Real-Time Database Workshop Eindhoven, 23 February 1995

Technologies

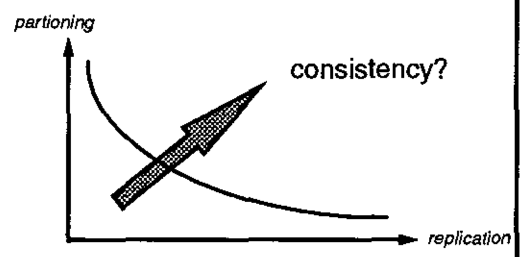
- performance
  - partioning to provide locall access
- reliability
  - replication to provide fault tolerance

L.J.M. Nieuwenhuis
The Netherlands

ptt research

Real-Time Database Workshop Eindhoven, 23 February 1995

Technologies



L.J.M. Nieuwenhuis
The Netherlands

ptt research

Real-Time Database Workshop Eindhoven, 23 February 1995

IN / TINA prototypes

- Relay Race
- INDIA

L.J.M. Nieuwenhuis
The Netherlands

ptt research

Real-Time Database Workshop Eindhoven, 23 February 1995

Relay Race

- primary-seconday copy model
- asynchronous
- 'weak consistency'
- stages: repository of update information
- read-only secondary copies

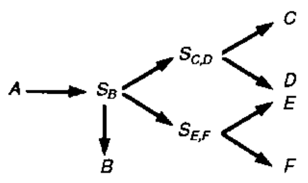
source: Cannataro and others, TINA Workshop 1994

L.J.M. Nieuwenhuis
The Netherlands

ptt research

Real-Time Database Workshop Eindhoven, 23 February 1995

Relay Race



$$\begin{tikzcd} & & & C \\ & & S_{C,D} \arrow[ur] \arrow[dr] & \\ A \arrow[r] & S_B \arrow[ur] \arrow[dr] \arrow[d] & & D \\ & B & S_{E,F} \arrow[r] \arrow[dr] & E \\ & & & F \end{tikzcd}$$

source: Cannataro and others, TINA Workshop 1994

L.J.M. Nieuwenhuis
The Netherlands

ptt research

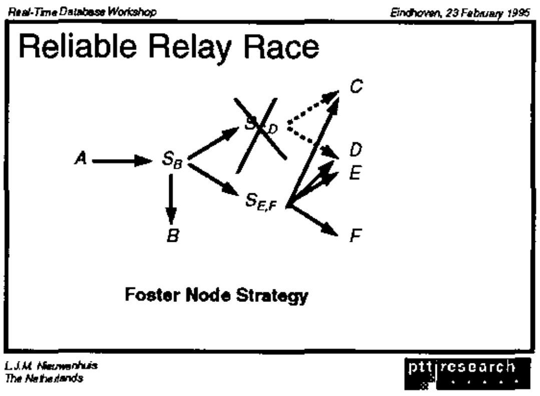
Real-Time Database Workshop Eindhoven, 23 February 1995

Reliable Relay Race

- Lack of fault tolerance
  - stage node failures
  - link failures
- Shadow Node strategy
  - redundant stage nodes are added
- Foster Node strategy
  - a stage services clients of other stage in case of failures

L.J.M. Nieuwenhuis
The Netherlands

ptt research

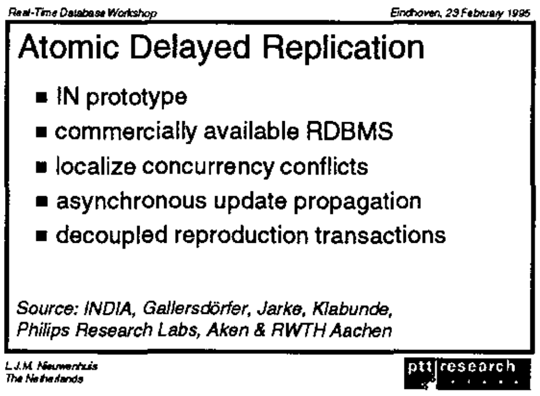


Real-Time Database Workshop *Eindhoven, 23 February 1995*

Reliable Relay Race

Foster Node Strategy

L.J.M. Nieuwenhuis
The Netherlands
ptt research



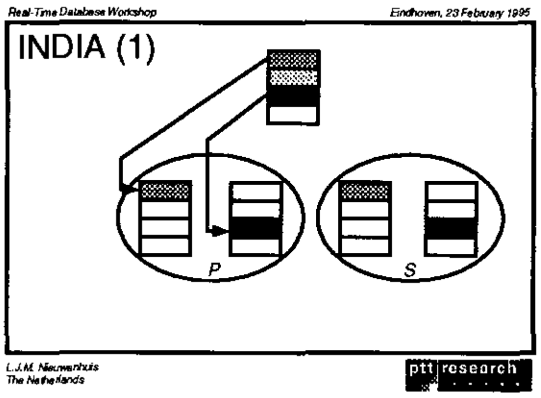
Real-Time Database Workshop *Eindhoven, 23 February 1995*

Atomic Delayed Replication

- IN prototype
- commercially available RDBMS
- localize concurrency conflicts
- asynchronous update propagation
- decoupled reproduction transactions

Source: INDIA, Gallersdörfer, Jarke, Klabunde, Philips Research Labs, Aken & RWTH Aachen

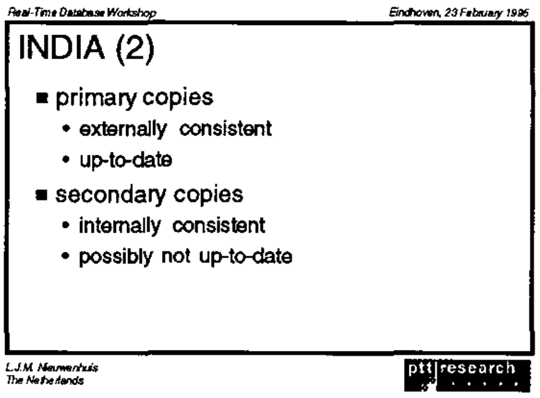
L.J.M. Nieuwenhuis
The Netherlands
ptt research



Real-Time Database Workshop *Eindhoven, 23 February 1995*

INDIA (1)

L.J.M. Nieuwenhuis
The Netherlands
ptt research

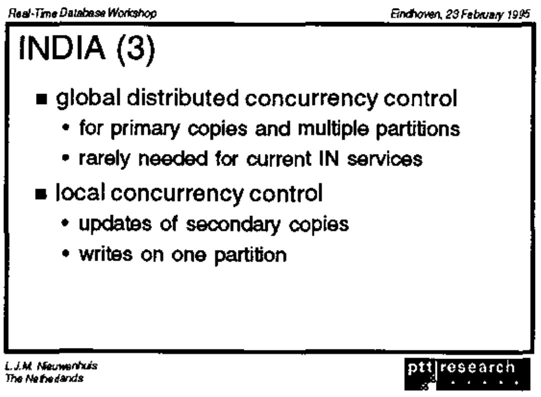


Real-Time Database Workshop *Eindhoven, 23 February 1995*

INDIA (2)

- primary copies
  - externally consistent
  - up-to-date
- secondary copies
  - internally consistent
  - possibly not up-to-date

L.J.M. Nieuwenhuis
The Netherlands
ptt research

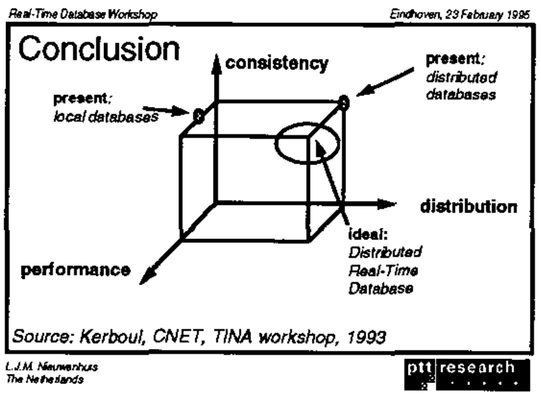


Real-Time Database Workshop *Eindhoven, 23 February 1995*

INDIA (3)

- global distributed concurrency control
  - for primary copies and multiple partitions
  - rarely needed for current IN services
- local concurrency control
  - updates of secondary copies
  - writes on one partition

L.J.M. Nieuwenhuis
The Netherlands
ptt research



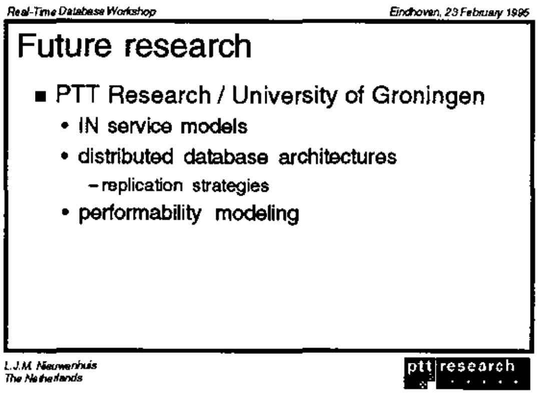
Real-Time Database Workshop *Eindhoven, 23 February 1995*

Conclusion

consistency
present: distributed databases
present: local databases
distribution
performance
ideal: Distributed Real-Time Database

Source: Kerboul, CNET, TINA workshop, 1993

L.J.M. Nieuwenhuis
The Netherlands
ptt research



Future research

- PTT Research / University of Groningen
  - IN service models
  - distributed database architectures
    - replication strategies
  - performability modeling

L.J.M. Nieuwenhuis
The Netherlands

ptt research

voorbereid - *prepared* ETM/RR Jan van der Meer	datum - *date* 1995-08-03	rev A	Dokumentnr. - *Document no.* ETM/RR-95:0021
doc. verantw./goedgekeurd - *Doc respons/Approved* ETM/RR	gecontr.- *checked*		dossier- *file*

Real Time Databases

Abstract

A short description is provided of our, Ericssons Intelligent Network Application Laboratory, view on the current possibilities of database technology. Especially aimed at the use for telecommunication applications.

The aim was not to be extensive, we consider ourselves users of databases and this document was written from that perspective.
Both inside and outside Ericsson other persons can be found who have much more detailed knowledge of the database-inside technology.

Jan van der Meer
Manager Ericsson Intelligent Networks Application Laboratory
etmjvdm@etm.ericsson.se

voorbereid - *prepared*	datum - *date* 1995-08-03	rev A	Dokumentnr. - *Document no.* ETM/RR-95:0021
doc. verantw./goedgekeurd - *Doc respons/Approved* / gecontr.- *checked*			dossier- *file*

1 GENERAL

In Applications for Telecommunications in general two types of databases can be found:

1 Handling real-time control information. In this case the contents of the database is used to influence the setup of individual calls. Examples are: Location database for mobile phones, service profile information for freephone or premium rate subscribers.

2 Handling management information. This can range from data related to the network setup, node configuration, subscriber base etc. to actual billing information.

Regarding retrieval speed and transaction capacity, the first type has received most attention sofar. In the near future we expect that also the databases providing data for management purposes, will be required to handle increased retrieval speed and transaction capacity. The possibility to access the current billing status of an individual user, can be an interesting service for users and operators.

Below the current status of technology, seen from an Ericsson perspective, is briefly described. Followed by some more information on the two general database types, and then the requirements to be put on database development.

2 Current technology status

Ericsson uses commercially available database systems in our telecom management systems. It was felt that by this approach the latest state of technology could be accessed. Although the systems perform as can be expected, we have problems with the provided capacity both in access speed and size.

For our Intelligent Networks and Mobile products we have developed our own databases. For both products access speed and reliability were important requirements. These databases were developed specifically for telecom applications, and geared to provide maximum performance on the Ericsson proprietary processing platform.

Our database implementations today handle 200 (for more complex queries) to 400 (for simple queries) transactions/sec. Access times are < 10 ms.
The processing platform is duplicated, clock synchronous system where both sides execute the same code continuously. Thus switchover in case of problems occurs without any impact on performance of the system, awareness of the application and loss of any telephone call.

The data is backed up on disk, where the backup frequency can be specified. In general the following approach is taken:

1 General backup, one or two times a day

2 Important data, dumped once every hour

3 Data changes are logged between the general backup dumps.

The dumps are stored in order where, when a reload is required, normally the most recent is selected. The operator can influence this selection, e.g. when software modifications have taken place.
The loading of the logged data-changes takes place under operator control, this to prevent data that lead to problems from being loaded.

Ericsson is currently analysing how to handle the data for telecom applications. Distributed databases are the most obvious answer. This technology is however not yet mature enough.

3 Requirements

For telecom applications we see the following requirements:

1 Access times should be <100 ms for 95% of the queries, if the time exceeds 200ms the transaction can be aborted.

2 If we assume that one record is required per user and/or terminal we come to database sizes of >2.000.000 records.

3 The records should appear to be of variable size. It must be possible to (dis)connect users to services and handle the appropriate user-specific service data.
Users will have a large amount of control over their service-data.

4 Security, apart from the obvious (one user cannot access data of another user) different service providers are not allowed to access data of users or services outside their domain.
One specific user could subscribe to services of different providers. The network operator will not be allowed to access the service provider specific user or service data.

5 Reliability, never lose anything.

We think that only a distributed database system can handle the requirement on size. There are also other reason to go for distribution e.g. reliability, access times, geographical structure of the network.

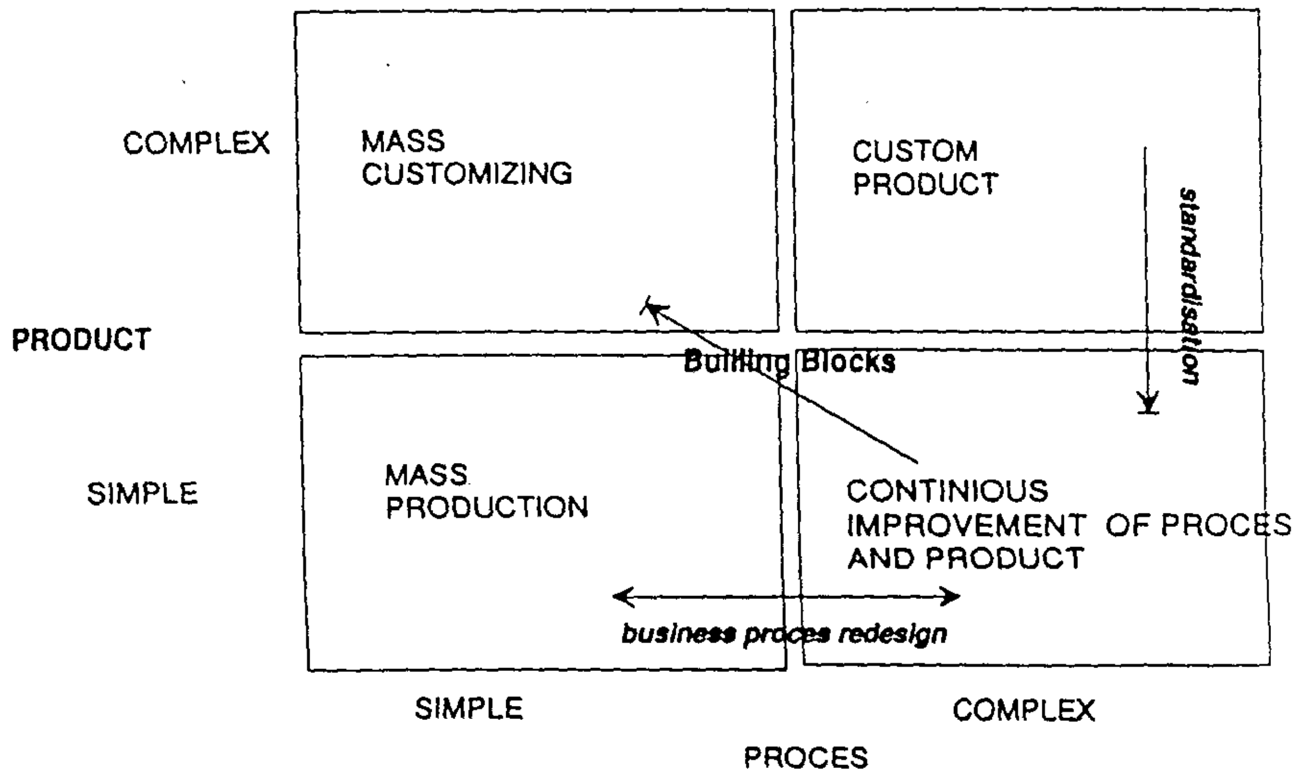
The above list of requirements should not be considered as being complete, it is mainly included here to get some 'feeling' of what is needed.

Database requirements for on-line financial transactions

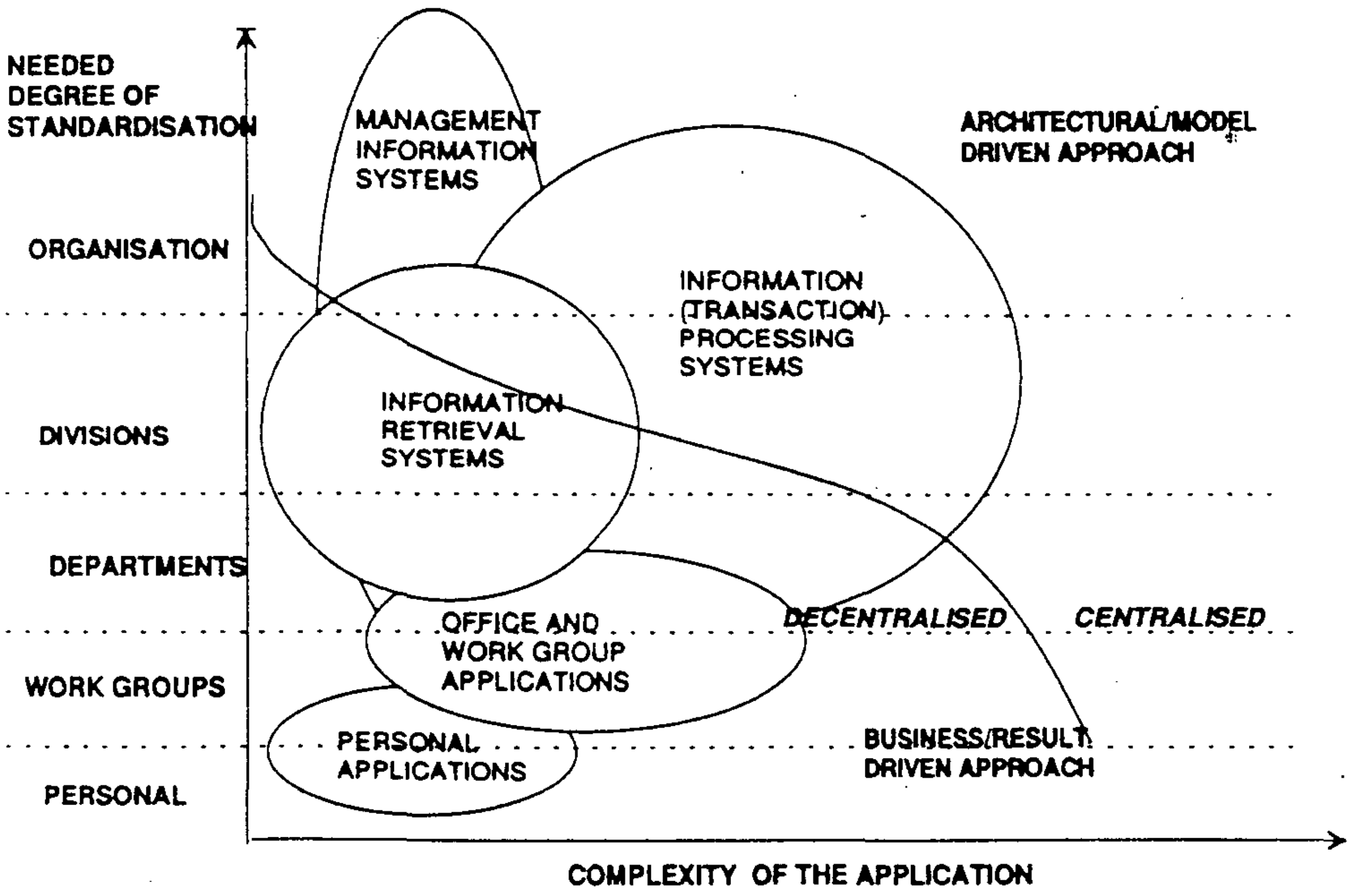
A.A. Vreven

Rabofacet, Zeist

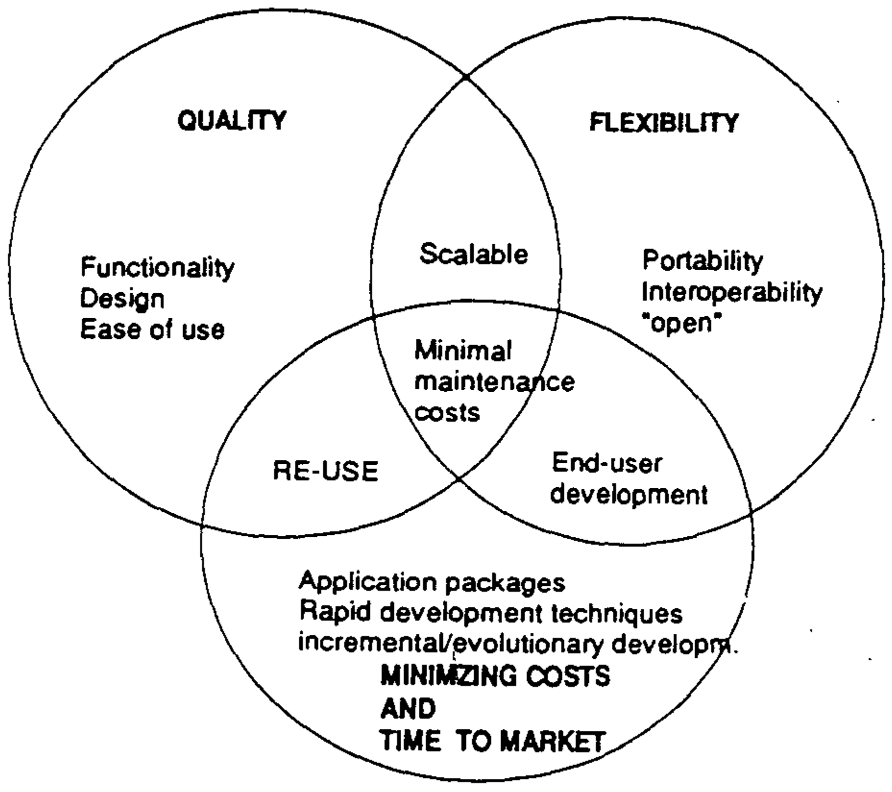
Time to market



Infrastructure

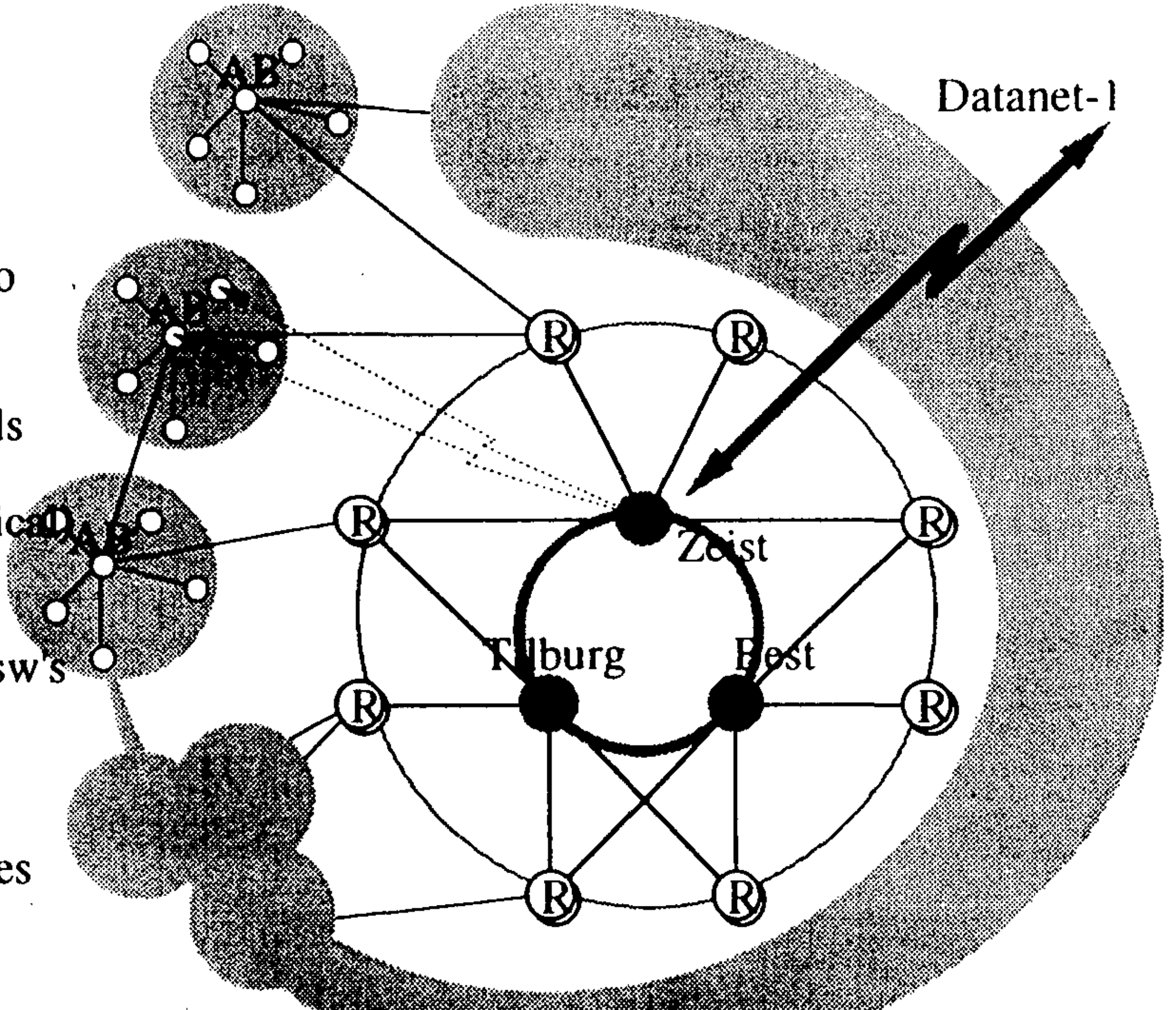

NEEDED DEGREE OF STANDARDISATION
ORGANISATION
DIVISIONS
DEPARTMENTS
WORK GROUPS
PERSONAL
MANAGEMENT INFORMATION SYSTEMS
INFORMATION (TRANSACTION) PROCESSING SYSTEMS
INFORMATION RETRIEVAL SYSTEMS
OFFICE AND WORK GROUP APPLICATIONS
PERSONAL APPLICATIONS
ARCHITECTURAL/MODEL DRIVEN APPROACH
DECENTRALISED
CENTRALISED
BUSINESS/RESULT DRIVEN APPROACH
COMPLEXITY OF THE APPLICATION

Flexibility



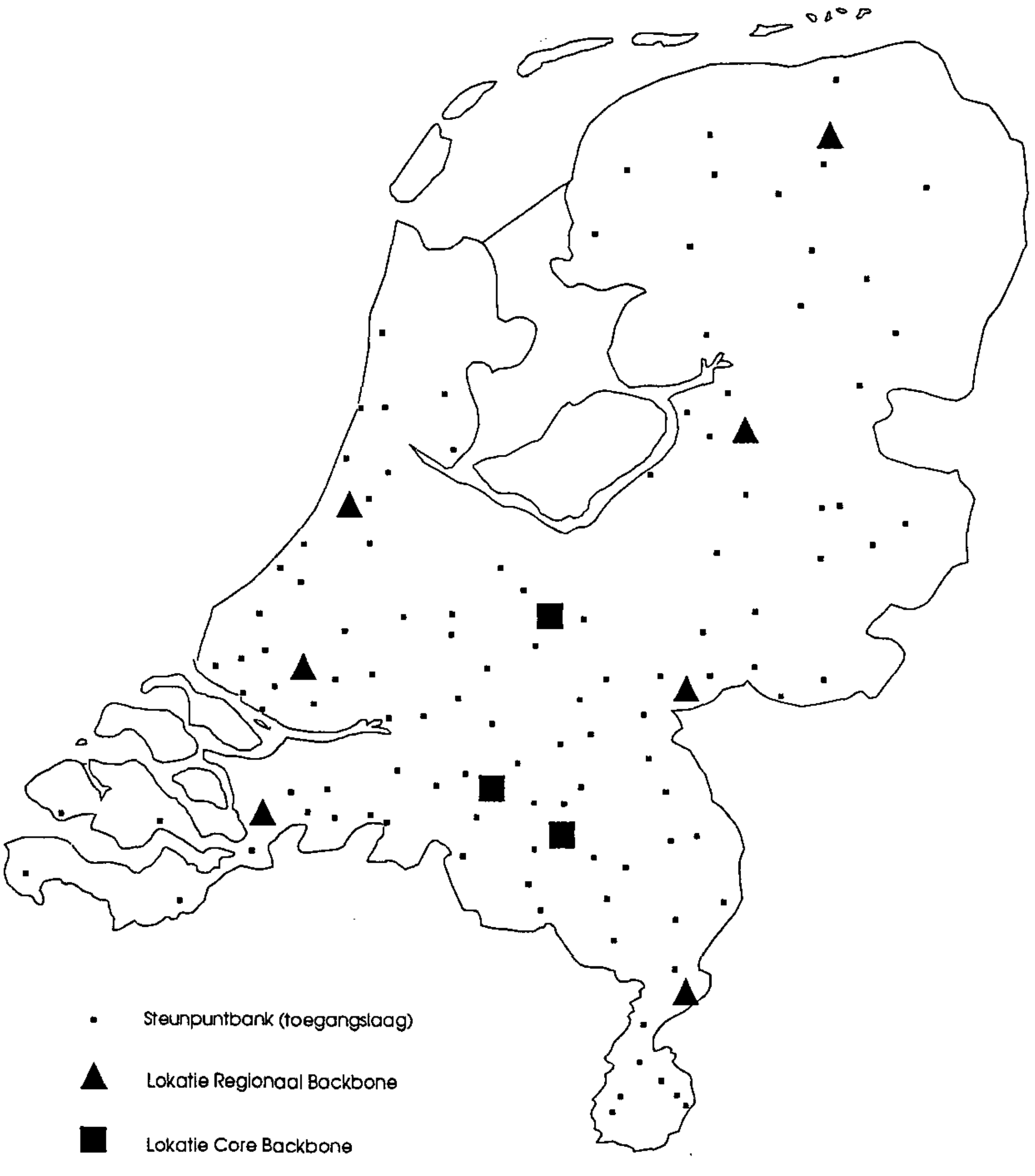
RTN95 topology

- three layer network
- meshed network on two levels
- redundancy on *all* levels
- within 98% of (theoretical) optimal topology:
  6 - 8 regional switches
  100 - 120 subregional sw's
- coupling met DN-1
- two management centres
- tenfold capacity



- *ring 1: multitude of 64 Kb or 2 Mb lines*
- *ring 2: 64 Kb lines*
- *ring 3: 64 Kb or 14400 bps*

RTN Toegangslaag



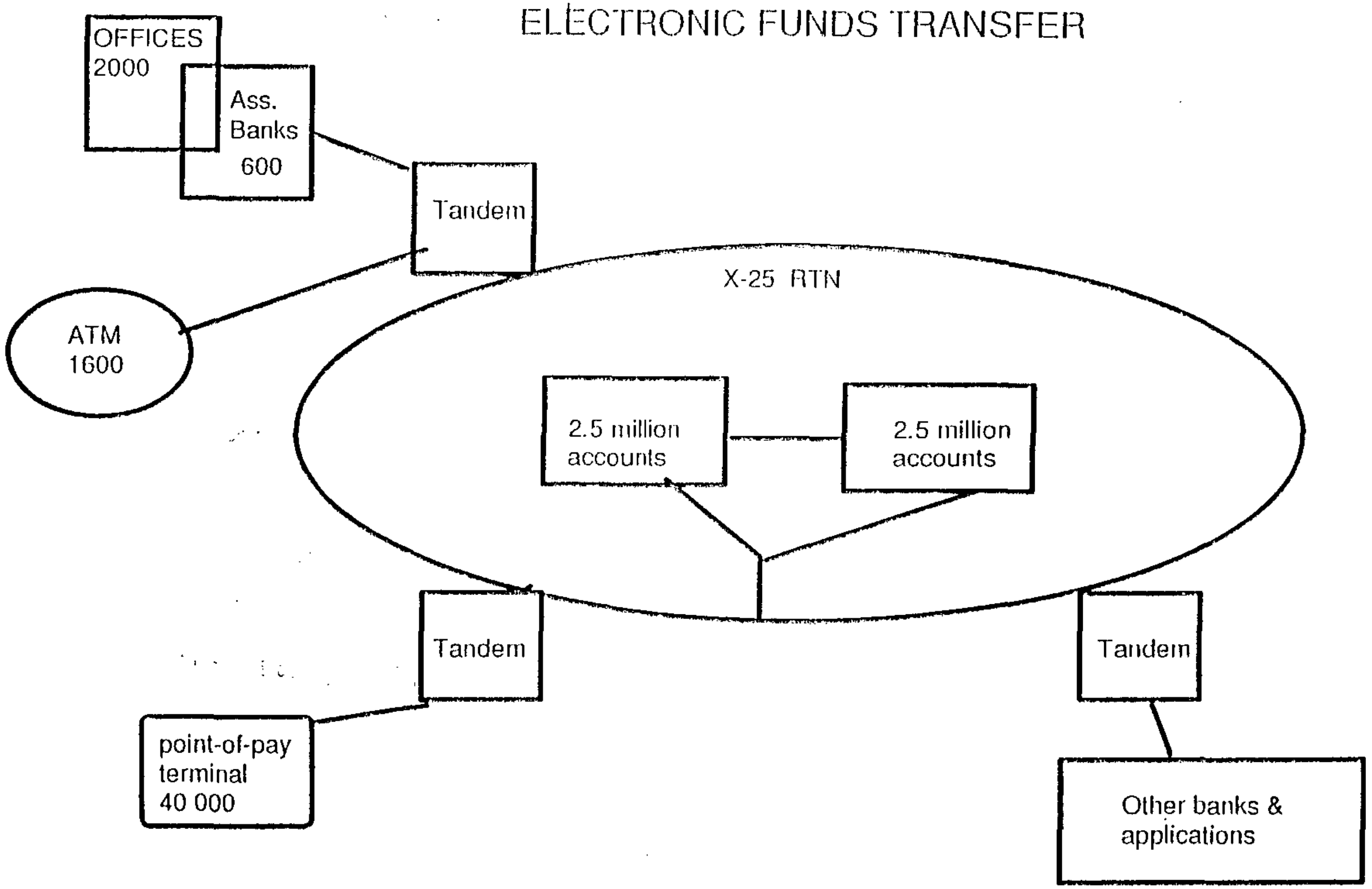
● Steunpuntbank (toegangslaag)

▲ Lokatie Regionaal Backbone

■ Lokatie Core Backbone

bron: RTN

ELECTRONIC FUNDS TRANSFER



Condition de faisabilité et temps de réponse maximum pour un système Transactionnel Réparti Temps Réel

Laurent George, Pascale Minet
INRIA, BP 105, Rocquencourt, 78153 Le Chesnay Cedex
Laurent.George@inria.fr, Pascale.Minet@inria.fr

RTS'95, Paris, Janvier 1995.

Résumé :

Les systèmes transactionnels répartis temps réel avec mise à jour en ligne de données réparties posent le triple problème de sérialisabilité/sûreté de fonctionnement/ ponctualité. Cet article utilise la théorie des jeux pour déterminer les conditions de faisabilité pour de tels systèmes. Ainsi le concepteur d'un système peut savoir à l'avance si un jeu de transaction satisfera ses échéances, il peut également obtenir un dimensionnement correct des files d'attente du système.

Introduction

La conception d'applications réparties temps réel doit pouvoir être prouvée correcte avant même d'avoir été implémentée [LELA94]. L'absence de conception prouvée correcte est à l'origine d'échecs industriels retentissants. Exemple: l'abandon du sytème de gestion de données de la station spatiale Freedom a coûté 500 millions de dollars.

Cet article montre comment déterminer le temps de réponse maximum pour un système transactionnel réparti temps réel. La connaissance de ce temps de réponse maximum permet de dire si un jeu de transactions respectera ou non ses échéances. Dans la première partie, nous définissons le modèle retenu et résumons brièvement l'état de l'art. Dans la seconde partie, nous explicitons les hypothèses adoptées. La solution algorithmique permettant de satisfaire simultanément les propriétés de sérialisabilité/ sûreté de fonctionnement/ponctualité est présentée. La dernière partie est consacrée à l'expression du pire cas, pour des hypothèses de charge données, permettant d'énoncer les conditions de faisabilité pour un jeu de transactions donné. Les conditions de faisabilité expriment le fait que les transactions doivent avoir terminé leur exécution avant leur échéance.

1. Problématique

Les systèmes transactionnels répartis temps réel avec mise à jour en ligne de données réparties posent le triple problème de sérialisabilité/sûreté de fonctionnement/ ponctualité.

En effet, les transactions interfèrent en lecture/écriture à des instants non prévisibles, la propriété de **sérialisabilité** (équivalence à une exécution sérielle) [BHG87] est donc suffisante pour maintenir la cohérence des données. De plus, dans un système temps réel, les transactions doivent se terminer avant leur échéance, la propriété de **ponctualité** (respect des échéances) doit donc être satisfaite. Comme dans tout système, il peut survenir des défaillances (crash, omission...), la propriété de **sûreté de fonctionnement** (propriété d'un système permettant à ses utilisateurs de placer une confiance justifiée dans le service qu'il délivre) [OFTA94] est donc souhaitée.

1.1. Modèle retenu

Le modèle transactionnel retenu se compose de clients et de serveurs interconnectés par un réseau de communication (voir Figure 1).

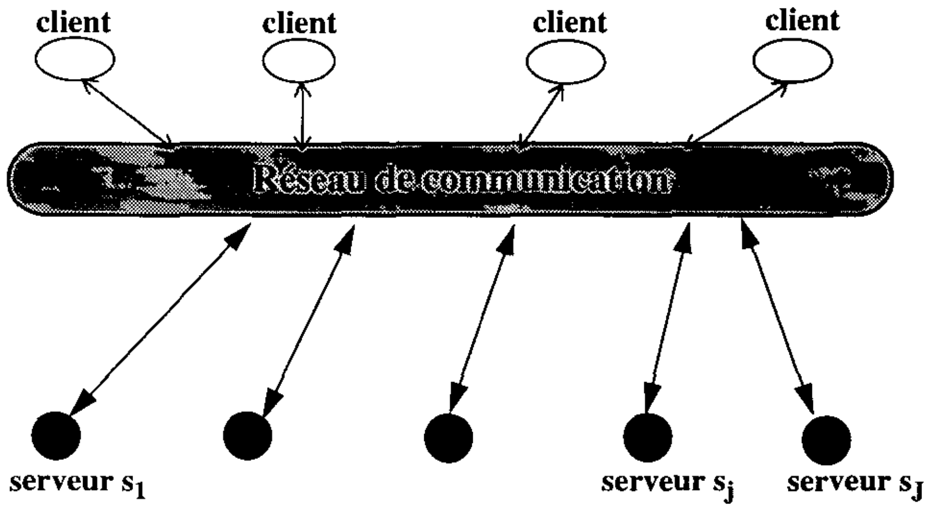


Figure 1 : Le modèle transactionnel retenu.

Un **client,** sur réception d'un stimulus extérieur ou à la demande d'un utilisateur, génère des transactions. Un **serveur** gère les objets (ressources locales rémanentes) dont il a la charge.

Une **transaction** est un graphe d'actions sans cycle, chaque **action** est un programme séquentiel exécutable sur un serveur, ce programme accède (lecture/écriture) aux objets gérés par ce serveur. A chaque transaction est associée une **échéance relative**. Une transaction d'échéance relative δ générée à l'instant t doit être terminée à l'instant $t+\delta$.

1.2. Objectif

L'objectif est d'une part de concevoir un **algorithme composite** permettant de satisfaire les trois propriétés de sérialisabilité / sûreté de fonctionnement / ponctualité. Cet algorithme composite aura :

- une composante contrôle de concurrence pour obtenir la sérialisabilité,
- une composante tolérance aux fautes pour obtenir la sûreté de fonctionnement,
- une composante ordonnancement pour obtenir la ponctualité.

Ces trois composantes doivent être **compatibles** pour garantir la vivacité (absence de blocage) et un comportement correct du système. A titre d'exemple, considérons deux transactions T1 et T2 d'échéance respective $\delta 1$ et $\delta 2$ et de priorité respective p1 et p2. Supposons de plus $\delta 1>\delta 2$, p1>p2 et T1,T2 accèdent à un même objet en mode conflictuel (exemple lecture/écriture, écriture/écriture). Un système utilisant un contrôle de concurrence favorisant dans un conflit la transaction avec la plus forte priorité (ici T1) et un ordonnancement **EDF,** Earliest Deadline First échéance la plus courte en premier (ex: EDF préemptif [LILA73]) favorisant T2 se bloquera: T2 ayant obtenu le CPU ne peut accéder à l'objet demandé, accordé à T1.

L'objectif est d'autre part de **dimensionner correctement** ces systèmes et prédire leur comportement en situation critique de charge élevée. Plus précisément, il est nécessaire :

- d'exprimer des preuves temporelles permettant de garantir les temps de service des transactions.
- d'établir des bornes supérieures sur les temps de réponse.
- d'exprimer des conditions de faisabilité et de stabilité.

1.3. Etat de l'art

Schématiquement, dans les systèmes répartis temps réel, nous pouvons distinguer deux approches dans l'état de l'art.

La première consiste à éviter les conflits entre transactions. Pour y parvenir, elle suppose une connaissance à priori des instants d'arrivée des transactions. Ces transactions sont générées périodiquement à des instants prédéterminés (voir par exemple [KOGR94], [BEMA94]). Ceci correspond à des hypothèses très fortes, hypothèses qui ne sauraient être acceptables dans tout système. Cette approche garantit la ponctualité et la sûreté de fonctionnement à condition que les hypothèses énoncées soient satisfaites en phase opérationnelle.

La deuxième approche consiste à résoudre les conflits pouvant survenir entre les transactions mais n'offre aucune garantie sur les temps de réponse. Par contre, elle garantit les propriétés de sérialisabilité et de sûreté de fonctionnement. Dans [JENS94] par exemple, l'ordonnancement utilisé est de type "Best Effort", il tend à maximiser le gain exprimé à l'aide de fonctions de valeurs temporelles définies par l'application.

L'objectif de cet article est de proposer une troisième approche plus générale puisqu'elle veut satisfaire simultanément les trois propriétés de sérialisabilité/sûreté de fonctionnement/ponctualité.

Pour garantir la ponctualité il faut recourir à un algorithme d'ordonnancement. En environnement monoprocesseur non préemptif, l'algorithme NP-EDF (Non Préemptive-Earliest Deadline First : échéance la plus courte en premier), a été prouvé optimal pour un scénario de tâches apériodiques [GMR94]. Ce qui n'est pas le cas de “Rate Monotonic” généralisé [SRS94].

2. Solution proposée

2.1. Hypothèses

Les hypothèses énoncées ci dessous sont nécessaires à l'expression du pire cas exposé dans le paragraphe 3.

• Au niveau des clients, seule la **densité maximale de génération des transactions** est connue. Elle s'exprime par la contrainte suivante :
Les clients ne peuvent pas générer plus de **N** transactions sur une fenêtre temporelle fixe de longueur **D**. Ce qui peut encore s'énoncer : sur une même fenêtre temporelle glissante de longueur D, nous ne devons pas avoir plus de 2N transactions générées.

soit to=instant de génération de la 1ère fenêtre

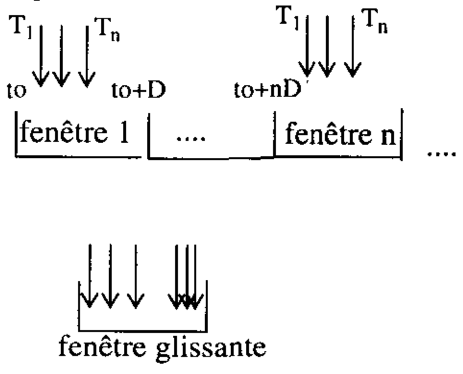


Figure 2 : densité maximale de génération des transactions

• Les **délais de transmission clients<->serveurs sont supposés bornés** et les bornes Min et Max sont connues.

• Au niveau d'une **transaction** sont connus **l'échéance relative** de la transaction et **son graphe**. Dans un but de simplification, nous limitons notre étude au cas des transactions dont le graphe comprend au plus une action par serveur. Dans le même but, les actions sont supposées indépendantes les unes des autres.

• Au niveau d'une **action**, la **durée d'exécution** est supposée connue ou tout au moins, nous connaissons une borne supérieure de cette durée d'exécution. Le **serveur** sur lequel va s'exécuter l'action, est connu.

• Le **nombre J de serveurs** est connu.

2.2. Solution algorithmique

2.2.1. Préliminaire

• Maintenir la sérialisabilité permet de garantir un ordonnancement cohérent des actions sur les serveurs. Or tous les serveurs n'ont pas la même connaissance des transactions en attente d'exécution. A la différence de [SRS94], la vue globale des transactions à ordonnancer n'est pas obtenue par recours à une entité centrale. Un consensus permet de garantir que tous les serveurs opérationnels partagent une même connaissance des transactions en attente d'exécution.

• L'ordonnancement des transactions doit permettre de garantir leurs échéances. Nous avons choisi une solution à base d'ordonnancement non préemptif NP-EDF. Cet ordonnancement est appliqué sur la liste des transactions résultat du consensus. Cela permet ainsi d'obtenir un ordonnancement distribué cohérent basé sur les échéances des transactions.

• La tolérance aux fautes nécessite généralement l'utilisation d'un consensus. Le protocole de consensus adopté dépend du type de fautes à tolérer (ex. crash, omission...) [DLS88] [HATO95].

En conclusion, la solution retenue est à base de consensus.

2.2.2. Principe

Lorsqu'un client génère une transaction, il diffuse aux serveurs concernés :
- l'identificateur de la transaction,
- l'échéance de la transaction,
- la liste des serveurs impliqués dans la transaction,
- la liste des actions qui devront être exécutées par ces serveurs (une action par serveur).

Les serveurs stockent les transactions reçues dans leur file *Non-ordo* des transactions non ordonnancées (voir Figure 3).

Un consensus permet aux serveurs de partager la même vue des transactions à ordonnancer. Chaque serveur extrait de sa file *Non-ordo* les transactions résultat du consensus. Il insère ensuite dans sa file *Ordo* les actions associées à ces transactions ordonnées selon NP-EDF et exécute ces actions.

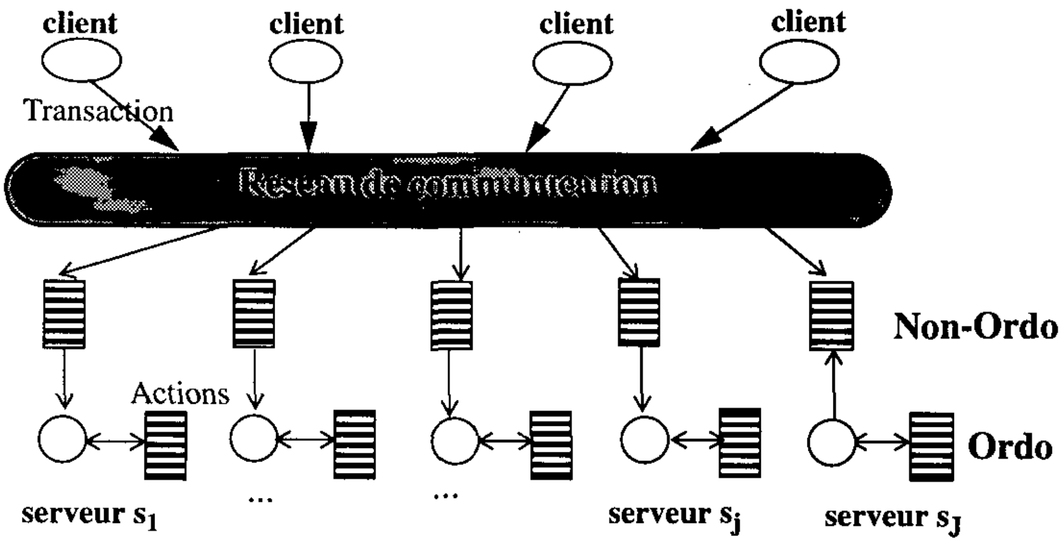


Figure 3: Principe de fonctionnement

2.2.3. Consensus

2.2.3.1. Principe du consensus

Informellement, un consensus entre les serveurs permet aux serveurs de partager la même vue des transactions à ordonnancer. Le terme "consensus" est utilisé ici dans une acception plus large que celle utilisée dans [HATO95], où les processeurs corrects décident d'une valeur proposée par l'un d'entre eux.

Le consensus se déroule selon le principe suivant:

- chaque serveur propose les transactions qu'il a reçues (i.e les transactions dans sa file Non-Ordo);
- le résultat du consensus est formé par l'union des transactions proposées par les serveurs.

Il suffit qu'un serveur ait reçu une transaction pour qu'elle soit proposée dans le prochain consensus.

2.2.3.2. Invocation du consensus

Le consensus est supposé de **durée connue C**. Le **délai inter-consensus** est au minimum de μ. Ce paramètre permet d'augmenter l'efficacité du consensus en augmentant le nombre de transactions proposées en une invocation. Mais du point de vue temps réel, ce paramètre augmente le temps de séjour d'une transaction dans la file Non-Ordo. Un compromis est donc nécessaire.

L'invocation sporadique du consensus est étudiée dans [AGLL94] où une transaction attend au plus un délai μ avant d'être proposée dans un consensus. La condition de stabilité est que toutes les transactions résultant du consensus σ_i sont terminées pour l'invocation du consensus σ_{i+1}.

Nous proposons dans cet article une adaptation de cette approche permettant d'obtenir la condition de stabilité la plus faible possible.

Le consensus σi est invoqué (voir Figure 4) dès lors que les trois conditions suivantes sont réunies:

- les transactions de σ_{i-1} ont terminé leur exécution.
- il s'est écoulé un délai $\geq \mu$ depuis l'invocation du consensus σ_{i-1}.
- il existe une transaction en attente d'ordonnancement.

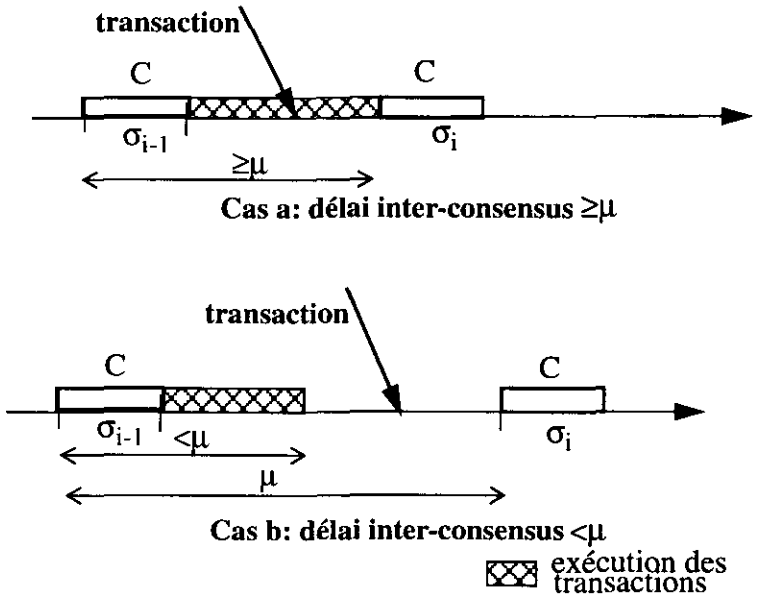

transaction
C
C
σ_{i-1}
σ_i
$\geq\mu$
Cas a: délai inter-consensus $\geq\mu$
transaction
C
C
σ_{i-1}
$<\mu$
σ_i
μ
Cas b: délai inter-consensus $<\mu$
exécution des transactions

Figure 4: Invocation du consensus.

3. Expression du pire cas

Le temps de réponse d'une transaction T_i, générée dans une fenêtre f, est déterminé par :

- $V_i(f)$, le temps d'attente de T_i avant d'être ordonnançée (temps d'attente avant consensus) : ce temps d'attente est maximisé lorsque la transaction arrive juste après l'invocation du consensus, la transaction doit alors attendre le consensus suivant.
- $X_i(f)$, le temps pour que T_i termine son exécution, T_i ayant fait l'objet d'un consensus : ce temps est maximum lorsque le consensus regroupe le plus grand nombre possible de transactions.

C'est pourquoi nous évaluons le nombre maximum de transactions pouvant appartenir au même consensus. Nous pouvons alors en déduire les conditions de faisabilité pour un jeu de transactions donné : les transactions doivent être exécutées avant leur échéance.

3.1. Consensus maximum

Considérons σ_m un consensus maximum en nombre de transactions. Par définition, ce consensus fait interférer le plus grand nombre possible de transactions, compte-tenu des hypothèses adoptées. Ce consensus est également maximum en nombre de fenêtres : une fenêtre est dite appartenir à un consensus σ s'il existe une transaction générée dans cette fenêtre appartenant à σ.

Soit to l'instant de génération de la première fenêtre temporelle (voir Figure 5).

Soit $to+(k+1)D-\theta_{1m}$ l'instant de génération de la première transaction de σ_m, avec k entier positif ou nul et θ_{1m} réel positif ou nul.

Soit $to+(k+w_m-1)D+\theta_{2m}$ l'instant de génération de la dernière transaction de σ_m, avec w_m=nombre de fenêtres participant au consensus σ_m et θ_{2m} réel positif ou nul. Nous avons de plus, $0\leq\theta_{1m}+\theta_{2m}<D$, sinon il serait possible de construire un comsensus regroupant plus de w_m fenêtres.

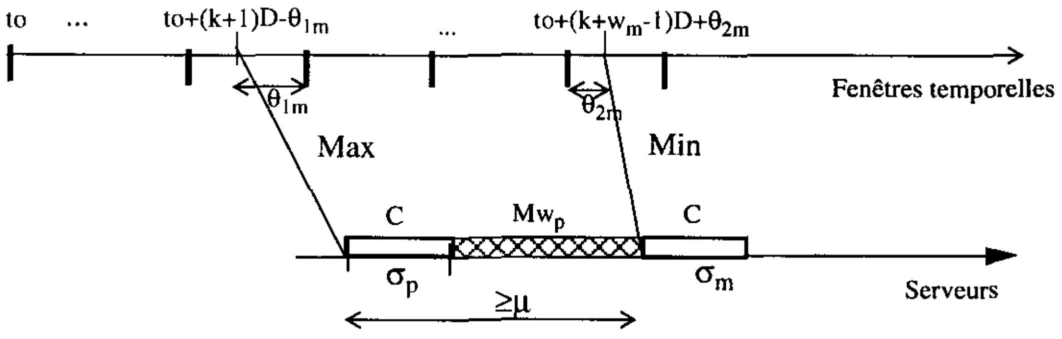


Figure 5: Consensus maximum

En pire cas, la première transaction met un délai Max et arrive à l'instant $t_{\sigma p}$ date d'invocation du consensus σ_p, consensus précédant σ_m. La dernière transaction met un délai Min et arrive à l'instant $t_{\sigma m}$ date d'invocation du consensus σ_m.

$t_{\sigma p}=to+(k+1)D-\theta_{1m}+Max$ et

$t_{\sigma m}=to+(k+w_m-1)D+\theta_{2m}+Min$.

Soit $M=max_{serveurs}$(durée d'exécution d'une fenêtre).

Remarque: dans le cas particulier où une transaction comprend exactement une action par serveur et où toutes les actions ont la même durée d'exécution x_0, $M=Nx_0$.

Nous distinguons deux cas, selon que la durée d'exécution des transactions de σ_p est supérieure ou inférieure à μ-C. Avec w_p=nombre de fenêtres participant au consensus σ_p.

- Premier cas: $C+Mw_p\geq\mu$. Nous avons alors, $t_{\sigma m}=t_{\sigma p}+C+Mw_p$, ce qui s'écrit encore :

  $to+(k+w_m-1)D+\theta_{2m}+Min=to+(k+1)D-\theta_{1m}+Max+C+Mw_p$.

  d'où $(w_m-2)D=\Delta+C+Mw_p-\theta_m$ avec Δ=Max-Min et $\theta_m=\theta_{1m}+\theta_{2m}$.

  Or w_m est un entier. Cet entier est maximum pour:

$$w_m = 2 + \left\lfloor \frac{\Delta + C + Mw_p}{D} \right\rfloor \text{et } \theta_m = \Delta + C + Mw_p - D\left\lfloor \frac{\Delta + C + Mw_p}{D} \right\rfloor$$

- Deuxième cas: $C+Mw_p<\mu$. Nous avons alors, $t_{\sigma m}=t_{\sigma p}+\mu$, ce qui s'écrit encore : $to+(k+w_m-1)D+\theta_{2m}+Min=to+(k+1)D-\theta_{1m}+Max+\mu$.
  d'où $(w_m-2)D=\Delta+\mu-\theta_m$ avec $\Delta=Max-Min$ et $\theta_m=\theta_{1m}+\theta_{2m}$.
  Or w_m est un entier. Cet entier est maximum pour:

$$w_m = 2 + \left\lfloor \frac{\Delta + \mu}{D} \right\rfloor \text{et } \theta_m = \Delta + \mu - D\left\lfloor \frac{\Delta + \mu}{D} \right\rfloor$$

D'où la formule générale donnant w_m et θ_m:

$$\boxed{w_m = 2 + \left\lfloor \frac{\Delta + max(\mu, C + Mw_p)}{D} \right\rfloor} \text{ et}$$

$$\boxed{\theta_m = \Delta + max(\mu, C + Mw_p) - D\left\lfloor \frac{\Delta + max(\mu, C + Mw_p)}{D} \right\rfloor}$$

$to+(k+1)D-\theta_m$ est l'instant de génération au plus tôt d'une transaction participant au consensus σ_m. De même $to+(k+w_m-1)D+\theta_m$ est l'instant de génération au plus tard d'une transaction participant au consensus σ_m.

Dans la suite de cet article, nous allons caractériser ce consensus maximum.

3.2. Calcul de wm.

Posons $w_0 = 2 + \left\lfloor \frac{\Delta + \mu}{D} \right\rfloor$ et $\theta_0 = \Delta + \mu - D\left\lfloor \frac{\Delta + \mu}{D} \right\rfloor$. Nous avons d'après les formules du paragraphe 3.1, $w_m \geq w_0$.
Nous distinguons deux cas selon la valeur de w_0:

- Premier cas: $C+Mw_0\leq\mu$

Lemme1: Si C+Mw0≤µ alors $\boxed{w_m = 2 + \left\lfloor \frac{\Delta + \mu}{D} \right\rfloor}$ et $\boxed{\theta_m = \Delta + \mu - D\left\lfloor \frac{\Delta + \mu}{D} \right\rfloor}$

Considérons le consensus initial w_1, w_1 est au plus égal à w_0, il ne satisfait donc pas $C+Mw_1>\mu$. Le plus grand consensus w_2 possible vaut w_0 car $C+Mw_1\leq\mu$.

Par récurrence, supposons $w_i \leq w_0$ et montrons $w_{i+1} \leq w_0$. Comme $w_i \leq w_0$, nous avons $C+Mw_i\leq\mu$, et donc le plus grand consensus w_{i+1} possible vaut w_0.

Donc $w_m=w_0$ est solution.

- Deuxième cas: $C+Mw_0>\mu$

Lemme2: Si C+Mwo>μ alors $\boxed{w_m \le \left\lfloor \frac{\Delta + C + 2D}{D-M} \right\rfloor}$ **et** $\boxed{w_m \ge w_0}$

Nous cherchons le plus grand $w_p \le w_m$ tel que $C+Mw_p \ge \mu$ et

$$w_m = 2 + \left\lfloor \frac{\Delta + C + Mw_p}{D} \right\rfloor$$

Or par définition $w_p \le w_m$. Nous en déduisons une borne supérieure sur w_m:

$w_m \le 2 + \left\lfloor \frac{\Delta + C + Mw_m}{D} \right\rfloor$ d'où $w_m \le 2 + \frac{\Delta + C + Mw_m - \theta_m}{D}$.

soit $w_m (D-M) \le 2D + \Delta + C - \theta_m$

or M<D (voir lemme 3) d'où $w_m \le \frac{\Delta + C + 2D - \theta_m}{D-M} \le \frac{\Delta + C + 2D}{D-M}$

d'où $w_m \le \left\lfloor \frac{\Delta + C + 2D}{D-M} \right\rfloor$

Une borne supérieure de w_m est donc $\left\lfloor \frac{\Delta + C + 2D}{D-M} \right\rfloor$.

Montrons que dans ces conditions $w_0 \le \left\lfloor \frac{\Delta + C + 2D}{D-M} \right\rfloor$

Pour ce faire, montrons que $\frac{\Delta + \mu + 2D}{D} < \frac{\Delta + C + 2D}{D-M}$.

Ce qui s'écrit aussi (car M<D):

$\mu D - M\Delta - M\mu - 2MD < CD$ soit encore: $D(\mu - C) < M(\Delta + \mu + 2D)$ (1)

or, $C + Mw_0 > \mu$ et donc en remplaçant w_0 par sa valeur nous avons:

$\left\lfloor \frac{\Delta + \mu + 2D}{D} \right\rfloor > \frac{\mu - C}{M}$ d'où, $\frac{\Delta + \mu + 2D}{D} > \frac{\mu - C}{M}$ ce qui s'écrit comme (1) qui devait être démontré.

Lemme3: La condition de stabilité est M<D.

Démontrons ce lemme.

- Premier cas : $C+Mw_0 \le \mu$.

Nous avons $M \le \frac{\mu - C}{2 + \left\lfloor \frac{\Delta + \mu}{D} \right\rfloor} < \frac{(\mu - C) D}{D + \Delta + \mu} < \frac{1 - \frac{\mu}{c}}{1 + \frac{D + \Delta}{\mu}} D < D$

• <u>Deuxième cas</u> : $C+Mw_0>\mu$.

La borne supérieure trouvée, ayant D-M en dénominateur, existe pour M<D. Ce qui s'exprime encore par le fait que la durée maximale d'exécution d'une fenêtre doit être inférieure strictement à la durée d'une fenêtre (condition nécessaire de non saturation des serveurs).

3.3. Calcul du temps de réponse maximum

Nous disposons donc de la valeur exacte de w_m et de θ_m lorsque $C+Mw_0\leq\mu$.

Nous disposons d'une borne supérieure de w_m lorsque $C+Mw_0>\mu$. Nous bornerons θ_m par D.

Ceci se résume comme suit, avec $w_0 = 2+\left\lfloor \frac{\Delta+\mu}{D} \right\rfloor$

$$\boxed{w_m = \begin{cases} 2+\left\lfloor \frac{\Delta+\mu}{D} \right\rfloor & \text{si } C+Mw_0\leq\mu \text{ et} \\ \left\lfloor \frac{\Delta+C+2D}{D-M} \right\rfloor & \text{sinon} \end{cases} \qquad \theta = \begin{cases} \Delta+\mu-D\left\lfloor \frac{\Delta+\mu}{D} \right\rfloor = \theta_0 \\ D \end{cases}}$$

Pour calculer le temps de réponse maximum d'une transaction, nous appliquons alors la méthode décrite dans [AGLL94] basée sur la théorie des jeux. Chaque transaction appartenant au consensus maximum est considérée isolément. L'adversaire va utiliser toutes les autres transactions du consensus maximum pour augmenter le temps de réponse de cette transaction. Pour toute transaction T_i, l'adversaire jouant contre T_i avec une transaction T_h va chercher à minimiser l'échéance absolue de T_h. Il s'en suit que T_h est générée au plus tôt dans sa fenêtre temporelle π, tandis que T_i est générée au plus tard dans sa fenêtre temporelle f. Pour exprimer le temps de réponse maximum de la transaction T_i, nous utilisons la fonction $F_i(h ; f, \pi)$. La fonction $F_i(h ; f, \pi)$ retourne 1 si la transaction T_h générée dans la fenêtre $\Pi \in [1..w_m]$ a une échéance absolue supérieure à l'échéance absolue de la transaction T_i générée dans la fenêtre $f \in [1..w_m]$. w_m et θ ont été définis ci-dessus.

```
F_i(h ; f, π) :
        si f = w_m
                si π= 1
                        si δ_h > (w_m - 2)D + θ + δ_i
                        retourne 1
                        fin si
                sinon
                        si δ_h > (w_m - π)D + θ + δ_i
                        retourne 1
                        fin si
                fin si
```

```
sinon
    si $\pi = 1$
        si $\delta_h > (f - 1)D + \theta + \delta_i$
        retourne 1
        fin si
    sinon
        si $\delta_h > (f - \pi + 1)D + \delta_i$
        retourne 1
        fin si
    fin si
fin si
retourne 0
```

En utilisant cette dernière fonction, il est alors possible d'obtenir une borne supérieure $r_{ij}(f)$ de la position en file d'attente Ordo d'une action de la transaction Ti sur le serveur s_j.

Soit $S(s_j)$ le nombre de transactions, parmi les N transactions d'une fenêtre, dont une action concerne le serveur s_j.

$$\boxed{\begin{array}{l} r_{ij}(f) = w_m S(s_j) - \displaystyle\sum_{\pi=1}^{w_m} \sum_{\substack{T_h \in \pi \\ Th \neq Ti \in f}} F_i(h;f,\pi) \quad \text{pour } S(s_j) \neq 0 \\ r_{ij}(f) = 0 \text{ pour } S(s_j) = 0 \end{array}}$$

On peut alors exprimer $X_i(f)$, avec x_0 désignant la durée d'exécution maximum d'une action.

$$\boxed{X_i(f) = Max_{serveursj}(r_{ij}(f)\, x_0)}$$

Remarque : Cette formule peut facilement être étendue au cas où la durée d'exécution d'une action est variable.

Intéressons-nous maintenant à $V_i(f)$, temps d'attente de T_i avant d'être prise dans un consensus. Ce temps d'attente est maximum lorsque T_i est générée en $t_i(f)$=début de la fenêtre f.

$t_i(f) = t_0 + (k+f - 1)D$, pour $f \neq 1$ et $t_i(f) = t_0 + (k+1)D - \theta_{1m}$ pour $f = 1$.

Le consensus σm se termine en

$t_0 + (k+1)D - \theta_{1m} + Max + \max(\mu, C + Mw_m) + C$.

D'où pour $f \neq 1$, nous avons : $V_i(f) = Max + \max(\mu, C+Mw_m) + C - (f - 2)D - \theta_{1m}$

ceci est maximum pour $\theta_{1m} = 0$.

Pour $f = 1$, nous avons : $V_i(1) = Max + \max(\mu, C+Mw_m) + C$. En conséquence :

$$\boxed{\begin{array}{l} V_i(f) = Max + \max(\mu, C+Mw_m) + C - \varphi(f-2)D, \\ \text{avec } \varphi = 0 \text{ pour } f = 1 \text{ et } \varphi = 1 \text{ pour } f \neq 1 . \end{array}}$$

Finalement B(i) désignant la borne supérieure du temps de réponse pour la transaction T_i, nous avons :

$$\boxed{B(i) = \max_{f \in [1, w_m]} \left[\max \; X_i(f) + V_i(f) \right]}$$

3.4. Exemples mumériques

J=8=nombre de serveurs,
N= 80 transactions par fenêtre,
D= 1 seconde= taille de la fenêtre,
C= 0,1 seconde= durée du consensus,
Max= 1seconde =délai de transmission maximum client<->serveur,
Min= 0,1seconde= délai de transmission minimum client<->serveur,
μ= 1 seconde= délai minimum inter-consensus.
Chaque transaction comprend une action sur chaque serveur.
Chaque action a une durée d'exécution maximale x_0.

3.4.1. Premier exemple $C+Mw_0<\mu$

x_0=3ms= durée maximum d'exécution d'une action sur un serveur.
Les transactions doivent respecter les échéances relatives suivantes :

de T1 à T20:	2,7 secondes,
de T21 à T40:	2,9 secondes,
de T41 à T80:	3,1 secondes

Nous avons alors $M=Nx_0$=0,240 secondes

$w_0 = 2 + \left\lfloor \frac{\Delta + \mu}{D} \right\rfloor$ ce qui donne w_0=3

$\theta_0 = \Delta + \mu - D\left\lfloor \frac{\Delta + \mu}{D} \right\rfloor$ ce qui donne θ_0=0,9 seconde

Nous vérifions la condition $C+Mw_0<\mu$
d'où w_m= 3 et θ_m= 0,9 seconde. Le consensus maximum regroupe 3 fenêtres et donc 240 transactions.

Toutes les transactions vérifient leurs échéances. Le jeu de transactions fourni est donc faisable. A titre d'exemple, le tableau suivant (voir tableau 1) donne le pire rang d'une action de la la transaction T1 dans la file Ordo en fonction de la fenêtre de génération.

Tableau 1: Rang de T1 dans Ordo

fenêtre	rang
1	100
2	180
3	240

3.4.2. Deuxième exemple $C+Mw_0>\mu$

x_0=5ms= durée maximum d'exécution d'une action sur un serveur.

Les transactions doivent respecter les échéances relatives suivantes :

de T1 à T20: 4,2 secondes,
de T21 à T40: 4,4 secondes,
de T41 à T80: 4,6 secondes

Nous avons toujours $w_0=3$ et $\theta_0=0,9$ seconde.
La valeur de M devient $M=Nx_0=0,4$ seconde.
La condition $C+Mw_0<\mu$ n'est plus vérifiée.
La formule donnant w_m est alors

$w_m = \left\lfloor \frac{\Delta + C + 2D}{D - M} \right\rfloor$ ce qui donne $w_m = 5$.

Nous avons $\theta_m=1$ seconde.
Le consensus maximum regroupe 5 fenêtres et donc 400 transactions.

Toutes les transactions vérifient leurs échéances. Le jeu de transactions fourni est donc faisable. A titre d'exemple, le tableau suivant donne le pire rang d'une action de la transaction T1 dans la file Ordo en fonction de la fenêtre de génération.

Tableau 2: Rang de T1 dans Ordo

fenêtre	rang
1	100
2	180
3	260
4	340
5	400

Conclusion

Les systèmes transactionnels répartis temps réel sont par nature complexes. L'approche présentée dans cet article permet de dimensionner correctement un système transactionnel réparti temps réel. De plus, elle permet à un concepteur de déterminer avant implémentation, la faisabilité d'un jeu de transactions donné. Les trois propriétés de sérialisabilité/sûreté de fonctionnement/ponctualité sont alors garanties.

Références

[AGLL94] E. Anceaume, L. George et G. Le Lann, "Timeliness proofs for real-time distributed transactional systems in the presence of high loads", Rapport de Recherche INRIA, Décembre 1994.

[BEMA94] A. Bertossi and L. Mancini, "Scheduling algorithms for fault-tolerance in hard real-time systems", Real-Time Systems, Vol7, N3, pp229-245, 1994.

[BHG87] P. Bernstein, V. Hadzilacos, N. Goodman, "Concurrency control and recovery in database systems", Addison-Wesley pub., 1987, 390 p.

[DLS88] C. Dwork, N. Lynch and L. Stockmeyer, "Consensus in the presence of partial synchrony", Jal of the ACM, Vol 35, No 2, April 1988, pp288-323.

[GMR94] L. George, P. Muhlethaler and N. Rivierre, "Optimality and non-preemptive scheduling revisited", Rapport de Recherche INRIA, Décembre 1994.

[HATO95] V. Hadzilacos, S. Toueg, "Fault-tolerant broadcasts and consensus", RTS95, Paris, January 1995, 9p.

[JENS94] D. Jensen, "A new generation open-family of scaleable real-time computer systems", RTS94, Paris, January 1994, pp207-221.

[KOGR94] H. Kopetz and G. Grunsteidl, "TTP a protocol for fault-tolerant real-time systems", Computer, IEEE, January 1994, pp.14-23.

[LELA94] G. Le Lann, "Certifiable critical complex computing systems", 13th IFIP Congress, Hamburg, Germany, August 1994.

[LILA73] C.L. Liu and J.W. Layland, "Scheduling algorithms for multiprogramming in a hard real time environment", Journal of the ACM Vol. 20, No. 1, pp. 46-61, January 1973.

[OFTA94] OFTA, "Infomatique tolérante aux fautes", Arago15, édité par MASSON, Février 1994, 226 p..

[SRS94] L. Sha, R. Rajkumar and S. Sathaye, "Generalized rate-monotonic scheduling theory : a framework for developing real-time systems", Proceedings of the IEEE, Vol82, N1, January 1994, pp.68-82.

On-line event reconstruction using a parallel in-memory database

Erco Argante
CERN, Switzerland / Eindhoven University of Technology, the Netherlands
(erco@harmony.cern.ch, tel. +41227674915)
Peter van der Stok
Eindhoven University of Technology, the Netherlands
Ian Willers
CERN, Switzerland

April 1995

PORS (parallel on-line reconstruction system) is an on-line event reconstruction system which uses the event reconstruction program of the CPLEAR high energy physics (HEP) experiment at CERN. Central to the system is a parallel in-memory database. This database is used as communication medium between parallel processes, which is possible because of the high performance of the parallel in-memory database. The database is applied to implement a farming strategy providing high CPU utilization. Farming is a simple example of one of many communication structures which can be acquired by using the in-memory database. The database provides structured storage of data with a short life time. PORS serves as a case study for the construction of a methodology on how to apply fast parallel in-memory databases and database techniques to HEP software, providing easier and better structuring of HEP data, simpler development, maintainability and reusability of HEP software, and straightforward parallelization of (existing) HEP software. PORS runs on a SPARCcenter 2000 8-node shared memory computer and reconstructs events at 117 Hz.

1 Introduction

Parallel in-memory databases offer a powerful means of communication, capable of facilitating parallelization of software, while providing high performance. In addition, they facilitate the structuring of data, and allow a separation of logical data presentation from physical data representation.

PORS is a case study which can perform on-line parallel event reconstruction using the event reconstruction program of the CPLEAR experiment at CERN, the European Laboratory for Particle Physics in Geneva, Switzerland. This is accomplished using a parallel in-memory database.

Event reconstruction is the process of converting raw detector data into interpretable physics results. Traditionally, event reconstruction is performed off-line, i.e. separate from the experiment. PORS can perform on-line event reconstruction, i.e. reconstruction during the experiment.

The main objective is to construct a methodology on how to apply databases and database techniques to (existing) HEP software, in order to:

- Acquire high performance by easily and efficiently applying the power of parallel computers via database techniques.
- Structure HEP data in an easy and consistent way by using the entity-relationship (E-R) model ([5]). HEP data comprises run-time data, i.e. the temporary data in a software program's memory space, as well as off-line data, i.e. permanent data stored on hard disk. Structured data facilitates development, maintainability and reusability of HEP software. Standardization of data structures via the E-R model results also in less software and hardware platform dependence.
- Parallelize (existing) HEP software with little effort. Parallel software should become less platform dependent and easier adaptable. This is achieved by using the parallel in-memory database as a powerful communication medium for parallel processes, offering much flexibility.

Within an in-memory database, the data reside in real memory in contrast with a disk-based database in which the data reside on hard disk. Former attempts to apply databases in on-line HEP applications failed due to a lack of performance. Parallel in-memory databases can provide this high performance.

This paper shows that the application of database techniques can lead to logically structured software which still has good performance. It will be argued that parallel in-memory databases are a viable alternative to current non-database software development techniques.

The lay-out of the paper is as follows. First, some general problems of HEP regarding computing are described. Relevant background of parallel databases is provided. This is followed by a description of the CPLEAR case study (section 4), and some of the design issues of the parallel in-memory database (section 5). The case study is evaluated, and advantages and disadvantages of applying databases are enumerated (section 6), in view of the problems mentioned before. Finally, some conclusions are drawn (section 7) and projections are given (section 8).

An extended version of this paper can be found in [8].

2 HEP software Problems

This section describes general problems seen in HEP experiments. These problems appear also in CPLEAR, the HEP experiment used in the case study.

Need for high performance

For new HEP applications and in particular the Large Hadron Collider (LHC) experiments at CERN, the estimated increase of computing power requirements for the coming ten years ranges up to three orders of magnitude depending on the area of application ([1]). Parallel processing will become an essential way to tackle this problem.

This paper shows that database techniques and parallel in-memory databases make it possible to efficiently and easily exploit the power of parallel computers for the type of applications presented here.

No uniform way of structuring data

Data are often stored in an unstructured or not uniformly structured way. This holds for run-time data as well as more permanently stored data. Data are often viewed and treated as binary objects without any defined structure. Permanent data are stored on sequential media, most commonly magnetic tape.

Software formats can be specific to an institute or even to a HEP experiment. An example is Zebra, a library to extend Fortran with more elaborate data structure facilities, which is specific to CERN ([6]). Another example is the OPAL ([7]) HEP experiment at CERN which has a set of reconstructed event data stored on hard disk, on which they defined their custom-made list structures to retrieve data. Also within a package like Zebra, there is a lack of enforced consistency. There is no uniform way of structuring the data, no general method how to model the data. Every programmer defines Zebra structures in an intuitive, personal way.

The E-R model, generally used in relational databases, helps to enforce structuring of data. Off-line data can be structured and stored in a disk-based database. Run-time data can be structured and stored in a parallel in-memory database. This results in data structures which are less hardware, software or application programmer dependent. Less hardware dependent, since the logical database structure is separated from the physical implementation. Less software dependent, since the resulting data will only be dependent on the E-R model, not on the software application from which they are coming. Less application programmer dependent, since application programmers are guided in their choice of data structures, which results in more uniformly modelled data.

A reference to the ADAMO ([10]) system should be made. In ADAMO, data structures can be modelled according to the E-R model via a data definition language (DDL). ADAMO provides a programming interface to C and Fortran. In this respect it does the same as the database presented here: it allows software programs to access data in a structured way with powerful statements (queries). In comparison to ADAMO, our system allows concurrent access and it can be used on a parallel computer.

Hardware dependence of high performance parallel programs

A problem in parallel software is that to obtain good performance, the machine architecture has to be taken into account ([2], [3]). Therefore, applications become machine dependent. Often the inter-communication pattern is fixed and explicitly defined. Changes in the hardware configuration lead to adaptations of the communication pattern.

Currently available tools, like communication paradigms PVM ([4]) or MPI, are still not capable to combine a good abstraction level with high performance ([3]). This reduces the software's maintainability and portability.

It is argued that a parallel in-memory database can be a powerful communication medium which provides abstraction without loosing much performance, and it also provides means for data distribution which PVM does not have. Therefore, it is a viable alternative to the currently available message passing tools.

Parallelizing existing software

Currently, many companies have much sequential software which is not parallelized, since they foresee that the gain does not compensate for the work and difficulties of parallelization ([2]). This situation especially holds for HEP environments. Farming of the sequential program is a solution which is not always satisfactory.

The database methodology helps tackling this problem. The database method conserves the original structure of an application. This is especially relevant for physicists, since they work on the reconstruction programs themselves. In the case study, this is accomplished as follows: the I/O statements are replaced by database statements. PVM is used in a similar manner. This method only works for farming. For more elaborate ways of coarse grain parallelism, like pipe-lining, the interface routines between the modules of the sequential program are replaced by database access routines. Synchronization between modules will be data driven.

A database provides data parallelism in an easy way, since data can be shared between processes while the database takes care of the data consistency.

3 Parallel in-memory databases

Relational databases

The purpose of a database is to store and retrieve data efficiently and conveniently. In a relational database, a collection of tables is used to represent data and relationships among the data. The SQL query language offers four types of access routines to retrieve information from a database and to change its contents: select, delete, insert and update. Indices are used to decrease search times through database tables. Logically associated access routines are gathered in transactions. A transaction is the unit of database action, i.e. it is atomic.

Parallel databases

A parallel database allows concurrent access. Parallel transactions can destroy database consistency. Concurrency control algorithms preserve database consistency in the context of parallel transactions ([5]).

In-memory databases

An in-memory database is a database in which the data reside in real memory. Parallel in-memory databases can be implemented on different hardware architectures, for example shared memory or distributed memory.

When comparing in-memory databases with disk-based databases, different design issues are relevant. For a disk-based database, disk access is by far the most expensive operation. Therefore, the major goal is to minimize this. For in-memory databases the access time on real memory is relatively small. Therefore, cost of maintaining database structures is a more relevant factor.

Data distribution and the E-R model: separation of concerns

In a parallel database implemented on a distributed memory computer, the data reside in the memories of multiple processing elements (PEs). Fetching data by a PE from another PE requires an inter-PE communication. Inter-PE communications are expensive operations in comparison to accesses to local PE memory. This makes data distribution an issue to minimize inter-PE communications and therefore increase performance.

A design rule is to completely separate the physical data distribution from the logical table structure of the database. So the data distribution is not reflected in the table structure; table structure is solely dependent on the E-R model. This separation of the logical view (i.e. the user view of the database) and the physical representation of the data in the database has two advantages:

- It facilitates portability and maintainability. It allows the implementor to change physical aspects of the database without changing the applications. For example, a database implemented on different types of hardware platforms will have the same interface to access the data.
- It makes the decisions about data modelling not dependent on physical aspects of the database which are hardware dependent. This helps to model the data in a uniform way, only dependent on the semantics of the data, and not dependent on details of implementation.

Minimizing inter-PE communications and separating logical view from physical representation are conflicting requirements which complicate the design of an in-memory database.

Transaction scheme and concurrency control

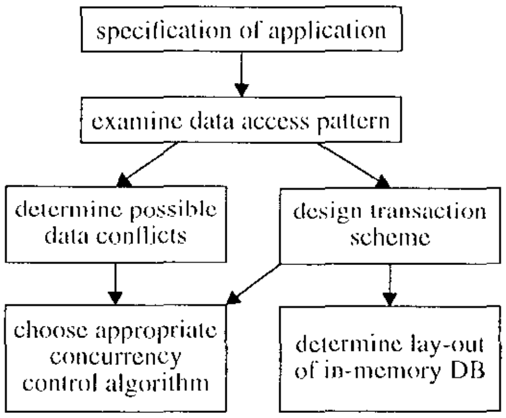
An application using the database, like PORS, consists of multiple processes which all access the database. The data access pattern of an application is the way the processes of the application access the database.

By examining this data access pattern of a process, a set of transactions is designed to retrieve the required data. The set of all transactions of all processes of an application is called the transaction scheme. The transaction scheme determines the lay-out of the in-memory database. This lay-out comprises table definitions, data distribution and process distribution.

Transactions of different processes may be executed in parallel. With this in mind, possible data conflicts are determined. The data conflicts are evaluated and an appropriate type of concurrency control is chosen to preserve data consistency. Advantage of this method of choosing concurrency control is that the overhead of the concurrency control will be low, since it does not have to deal with all possible data access patterns.

The steps described above form an important design strategy which supports the high performance of the database. Summarizing, the design strategy comprises that transaction scheme and type of concurrency control are optimized against the data access pattern of the application. In figure 1, the described steps are depicted. The steps should be carried out if a new application is to be used with the database in an optimal fashion. It is intended to distinguish classes of applications each for which a suitable type of concurrency control exists, thus avoiding that for every new application the complete diagram of figure 1 needs to be followed.

FIGURE 1. How to integrate an application with the in-memory database



specification of application
examine data access pattern
determine possible data conflicts
design transaction scheme
choose appropriate concurrency control algorithm
determine lay-out of in-memory DB

4 Case study: on-line event reconstruction for CPLEAR

The case study is an example of the integration of the application CPREAD, event reconstruction program for the CPLEAR HEP experiment, with the parallel in-memory database obtaining on-line parallel event reconstruction. The case study comprises the design and implementation of the database and the integration of the application with the database. It should be noticed that the parallel in-memory database is reusable; it can be used in conjunction with applications other than CPLEAR.

The case study is an implementation on a shared memory computer. It is a pilot system for the complete project with the full functionality not yet implemented. It therefore shows little of the potential functionality of the database. The implementation on a distributed memory machine will show more of its functionality, since data distribution becomes important, and the power of queries will be exploited. The case study is described in detail in [9].

CPLEAR & CPREAD

CPLEAR is a HEP experiment at CERN which investigates the C-P violation phenomenon. The experiment has run for a couple of years and about 100 people are involved. CPREAD is a 260k lines Fortran source code program which is used to reconstruct events produced by CPLEAR. About 100 Gbytes of data have to be reconstructed annually. At run-time, the program size is 13 Mbytes. CPREAD is often changed and adapted.

Description of the system in operation

Figure 2 shows the system. It is implemented on an 8-node shared memory computer. The dotted box comprises the database. The database together with the database library (*DB library*) form the reusable part of the system.

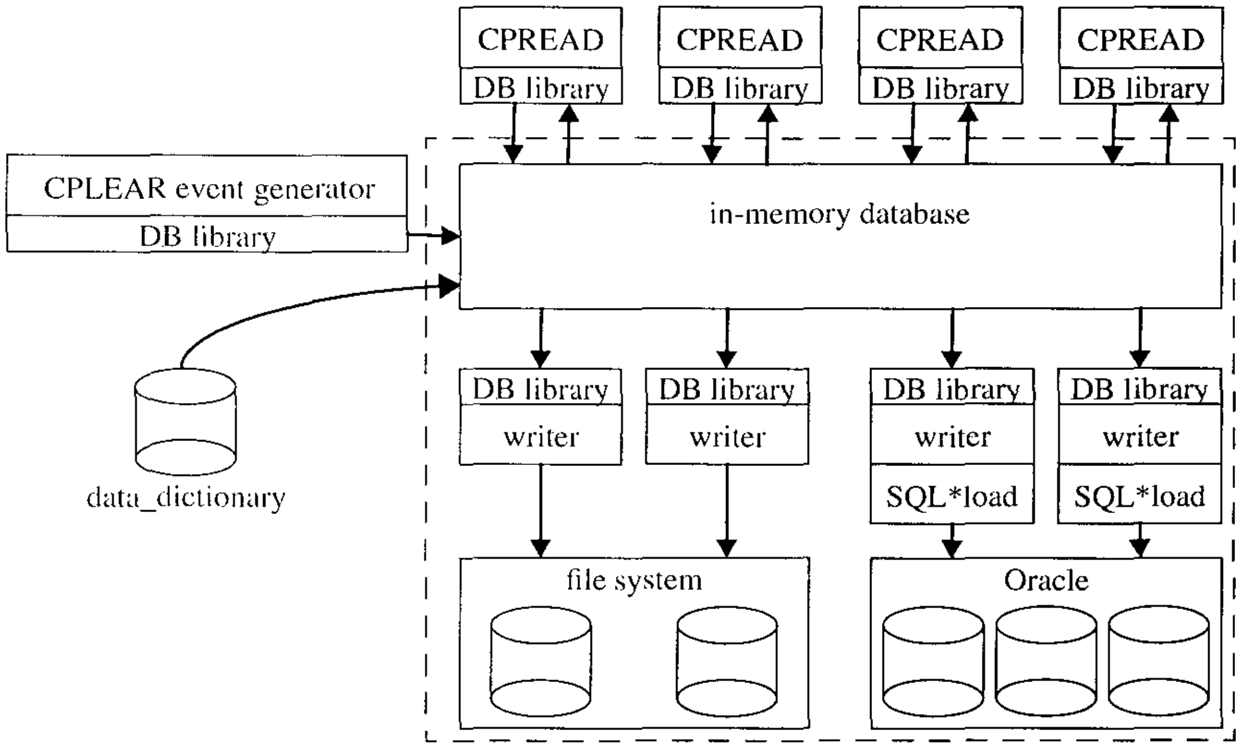
The *CPLEAR event generator* is a source of CPLEAR event data. It inserts Zebra ([6]) blocks with raw (i.e. non-reconstructed) events into a table, say table 1, of the in-memory database, at a specific rate. *CPREAD workers* (the farm workers) try to retrieve Zebra blocks with raw events from this table. If a *worker*'s request fails, because there are no data available, it restarts its request after a specific time. A *worker* reconstructs the events and reconstructed valid events together with reconstruction information are inserted into another table, say table 2, of the in-memory database. *Writers* try to retrieve accepted events from table 2 to write them to permanent storage. If a *writer*'s request fails, because there are no data available to be written to permanent storage, it restarts its request after a specific time.

How is it prevented that an event is retrieved multiple times by the *workers*? This is accomplished via the transaction scheme (see section 5). The transaction scheme provides data driven farming with high CPU utilization. Just by connecting *workers* to the database a farm is formed.

It should be noticed that there is no master in the system which controls the placement of data. The absence of a master in the database will enhance its scalability (see section 5).

Permanent storage can be a file system on hard disk or the Oracle 7.1 parallel database. In the case of a file system, the data are stored in one or more files. In the case of the Oracle database, the *SQL*loader* tool is used to insert

FIGURE 2. On-line event reconstruction for CPLEAR using a parallel in-memory database



data into the Oracle database. This tool, provided by Oracle, makes it possible to insert data into the Oracle database at a high rate. Only data with a long lifetime, i.e. reconstructed events, are stored on permanent storage. Data with a short life-time only reside in the in-memory database.

Each *worker* resides on its own processor. Since the overhead of the *CPLEAR event generator* and *writers* is relatively small in comparison to the load of a *CPREAD worker*, one *worker* for every processor of the machine gives the best performance. The load-balancing of processes over processors is taken care of by the operating system, or is done by the user explicitly.

At run-time, *workers* or *writers* can be added or removed from the system, i.e. the farm size can be changed dynamically. This does not lead to any data inconsistencies or data loss. It is a standard feature of the database: the database allows processes to connect or disconnect at run-time.

The in-memory database resides in memory which is shared by all processes accessing the database. *DB library* is a library linked to all processes accessing the in-memory database. It provides database access routines, concurrency control and connects the process to the database.

At start-up, a file called *data_dictionary* is used to build the database. It contains information about the lay-out of the database (table definitions, column definitions, etc.).

Hardware

The system is implemented on a SPARCcenter 2000 shared memory computer. It has 512 Mbytes shared memory and eight 40 Mhz SPARC processors with 2 Mbytes cache each. Cache memory is local to a processor. Off-cache means global to all processors.

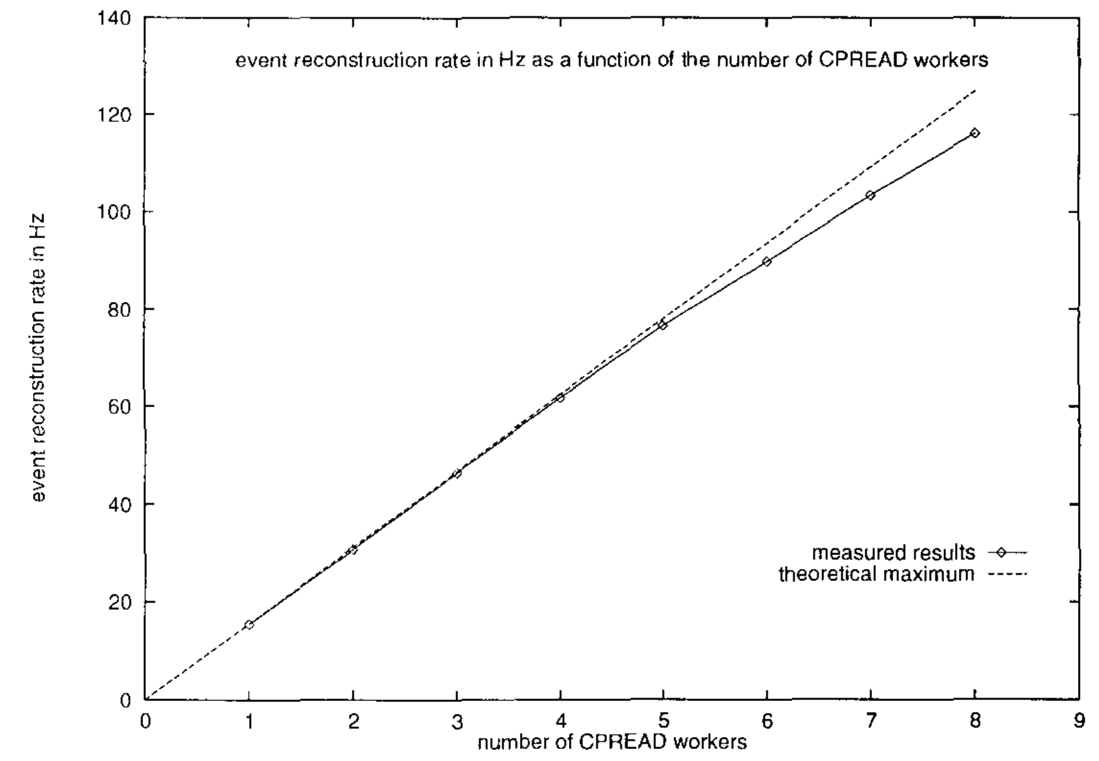
Performance

Multiple processes concurrently try to access the same table of the database. This table contention forces processes sometimes to wait. Administration of the database causes overhead. Executing access routines can cause searches. This overhead is relatively more important for an in-memory database than for a disk-based database, since the time to fetch the data items themselves is in the same order as accessing an index. Measurement of this overhead shows database performance.

This section gives some figures to show how the database approach performs in comparison to former non-database methods which have high performance, but are more hardware dependent.

Figure 3 shows the performance of PORS: the event reconstruction rate as function of the number of *CPREAD workers*. In this case, reconstructed data are written to files

FIGURE 3. Event reconstruction rate as function of the number of CPREAD workers



on disk. A standard stand-alone CPREAD worker, i.e. without using the database, can reconstruct events at a rate of 15.5 Hz on the SPARCcenter computer (in this case it uses one processor). So assuming that full scalability is possible, 8 processors would process events at 124 Hz. The system does process events at 117 Hz with 8 processors. So 7 Hz are lost by the overhead caused by the database. It should be kept in mind that when farming would be implemented without a DB, there would be an overhead also.

Some remarks on the figure with respect to scalability are given. These explain the decreased performance for the measurements with the higher number (= 6,7,8) of *CPREAD workers* in comparison to the theoretical case:

- The SPARCcenter has an automatic load balancer. That means that for the measurements with the lower number (= 1,2,3,4,5) of *CPREAD workers*, processes of other users, *writers* and *event generator* are running on the "free" processors, i.e. the processors where no farm worker runs. This means that a *CPREAD worker* has a processor for its own and performance degradation is only caused by waiting for database access.
  For the higher number of *CPREAD workers*, the processors are shared between *CPREAD farm workers*, *event generator*, *writers*, system processes, and processes of other users (although it was taken care for that none of the other users has put a heavy load on the machine during tests). This gives a skewed view, because the measurements with a higher number of *CPREAD workers* therefore show a relatively bad performance.
- For the higher number of *CPREAD workers* the SPARCcenter pages. This gives a skewed view since for the higher number of *workers* not just the database overhead is causing performance degradation, but also the hard disk activity caused by the paging.
- For the higher number of *CPREAD workers* the limited memory bandwidth of a shared memory machine might be visible.

Some typical execution times for access routines during run-time of PORS are provided. The results of the same tests without PORS running are equal. The provided timings are averages. This is for a table of two columns which contained about 100 rows. The first column has type integer, the second column has type binary and has a size of 23 Kbytes.

- select: 1 microsec.
  This select uses the index on the first column to perform the search. The select returns a pointer; so the copying of the data is not included.

- delete: 7 microsec.
  A delete also comprises an update of the indices, therefore it takes longer than a select.
- update: 0.11 millisec.
  This is an update of the first column of a row, i.e. an integer.
- insert: 3.2 millisec.
  The difference in time with the other routines comes from the fact that 23 Kbytes have to be moved from one memory location to another.
- memcopy of 23 Kbytes: 3.2 millisec.
  To get a better view on the performance of the insert routine, the time to perform a memory copy of 23 Kbytes is shown.

An insert routine comprises a memory copy. To explain the results for the insert, also the time a memory copy takes is measured. It can be seen that the time an insert takes is fully dependent on the time a memory copy takes; database overhead is not visible. This is a property of the SPARCcenter computer: copying global variables takes relatively long. Local variables can reside in the cache and therefore are copied much faster. The memory copy of a *local* variable of 23 Kbytes takes 1.5 millisec.

To obtain a better view on the database overhead, also measurements with smaller data items are carried out. The measurements are done for a table of one row with one integer column. For this type of table it is relatively simple to measure in-cache and off-cache results. Secondly, a table with small data items shows database effects more clearly, since the time to copy one integer to another memory location is insignificant.

• In-cache:	select:	1.0 microsec.
	delete:	2.1 microsec.
	update:	4.0 microsec.
	insert:	5.3 microsec.
• Off-cache:	select:	5 microsec.
	insert:	18 microsec.

A comparison of the in-cache figures for the different routines shows that they are consistent with the amount of administration that has to be done by the routine. These results show much clearer that database overhead is not big. Measurements of off-cache delete and off-cache update are not possible.

Eventual connection of the system to CPLEAR

At present, the system is not connected to the CPLEAR experiment to perform on-line event reconstruction. Necessary adaptations are the construction of an interface between experiment and system, and an increase of the permanent storage size.

5 Design aspects

Some interesting design aspects of the case study are discussed.

Transaction scheme

The transaction scheme is the set of all database transactions of all processes which together determine the execution flow of the parallel program. In the CPLEAR case study the transaction scheme serves, among others, the following purpose: how can it be avoided that a Zebra block is reconstructed multiple times.

An approach with counters is used, since it provides more concurrency than an approach where table locking is used. Zebra blocks are numbered by the generator in a consecutive increasing sequence by means of an identification number, i.e. a column "ID" is added. A *CPREAD worker* acquires an ID number from a counter (a table with one row of one column is used as a counter) and increments the counter by one. With the acquired ID the worker can retrieve the corresponding Zebra block. Since there is no other process which might access this Zebra block, no concurrency control on the data items is needed. The only required concurrency control is on the access of the counter. This means that table access is more concurrent. Access to the counter is purely sequential, but since accessing a counter takes only 5 microsec., this hardly influences scalability in a negative way.

At the output side of CPREAD, exactly the same procedure can be followed: a *writer* acquires an ID from a counter and retrieves the corresponding event to write it to disk. The accepted events are tagged by an ID number by the *CPREAD workers*. These ID numbers form a consecutively increasing sequence.

Distributed nature: no master

A design rule for the in-memory database is that the concept of a central master is avoided. So every process makes its own decisions. The advantage of this approach is that it enhances the scalability of the system, since a master can become a bottleneck.

The farming is data driven. This means that a *worker* asks for data when it is ready to process data. So there is no master which decides to which *worker* data are sent.

Transport of data to the right process is done via the transaction scheme. Every process takes its own decisions.

6 What do we gain?

By evaluating the case study, advantages and disadvantages of applying databases are enumerated; they are evaluated against the problems mentioned in section 2.

Performance requirements

A parallel in-memory database can give high performance. This performance allows the database to store data used by the experiment while it is running without becoming a bottleneck.

Avoid superfluous copying of data

Data can be shared between multiple processes instead of every process having its own local copy. Concurrency control takes care that data consistency is preserved. This offers an easy way to obtain data parallelism.

Incorporating a commercial database

By integrating a commercial database like Oracle into the system for off-line purposes, there are a lot of additional advantages:

- structured storage of permanent data
- conformance to industrial standards
- data representation to the user is platform independent
- network access among different platforms
- it offers tools (e.g. graphics or statistics tools) which are ready to use and which are directly applicable to the data

Communication paradigm

A parallel in-memory database can serve as a powerful communication paradigm. It can be seen as a layer which makes applications more soft- and hardware independent. In comparison to PVM, the database approach offers more power, because SQL queries can specify data requests more precisely than the primitives of PVM. This allows easy implementation of rather complex control structures, which in its turn provides flexibility. These control structures (like farming, buffering or monitoring system performance) do not have to be implemented explicitly. They are inherent to using a database. The database offers a data driven approach in which the availability of data controls execution of the parallel program. Control of parallelism of the application is obtained via the concurrency control algorithm which preserves data consistency. Examples are:

- Farming. The case study shows that for farming on a shared memory machine, the data driven approach takes care that CPU utilization is high. If there is a worker ready to receive data, and data is available, these data will be immediately available for that worker.
- Buffering. Databases provide flexible buffering which is easy to set-up and easy to change even at run-time. For example by specifying an appropriate data distribution on a distributed memory machine, data can be sent in advance: instead of retrieving the data from another PE at the moment they are really needed, the data can be stored on that PE beforehand. By using a database, this can be accomplished by just specifying a table to reside on a specific PE. In this context, a database should be viewed as memory space manager, rather than storage medium: it provides powerful tools to easily specify where data should reside.
- Monitoring. The database offers facilities to easily monitor the system. For example the data distribution among the PEs on a distributed memory machine can be monitored. Queries can answer questions like: "should more data be sent to some PE?".

Summarizing, by using a database flexibility is gained. Adapting a query takes less effort than changing the implementation of the application. Most features can be implemented without using a database, but they will be hard coded, or if flexible, the implementation takes a lot of effort. The database approach provides this flexibility implicitly.

7 Conclusions

The case study showed that by applying a database, control structures (data driven farming, dynamically changing farm size) were available without being explicitly implemented. Performance turned out to be quite scalable.

By connecting the workers to the database, and choosing an appropriate transaction scheme, data driven farming is established in which availability of the data controls the execution flow of the program. CPU utilization is high. Without being explicitly implemented, it is possible to dynamically change the farm size.

The case study left many features of the database unused. The coarseness of the data granularity did not show the strong points of the E-R approach and communication ability of the database. The implementation on a shared memory computer left features like data distribution in combination with non-uniform memory access untested and not revealed.

Performance. The on-line event reconstruction is able to reconstruct events at 117 Hz. This is only a loss of 7 Hz against the theoretical maximum of 124 Hz if full scalability is assumed. Database overhead turns out to be small. Properties of the SPARCcenter machine turn out to be important.

8 Future work

Further parallelization of the case study

The case study will be extended with parallel track fitting. Track fitting is a part of the reconstruction process performed by CPREAD in which particle tracks are calculated from the detector data. This part will be parallelized, i.e. there will be track-fit workers where each can perform a track-fit. Communication between track-fit workers and CPREAD workers will be done via the database.

This parallelization goes hand in hand with an increase of the data granularity to a sub-event level. The aim of this study is as follows:

- The increased parallelism should provide a reduced latency and less memory usage. The reduced memory usage comes from reduced code replication.
- It is a heavy test for the approach to use a database as a means of communication. The number of communications increases and the size of the communicated data packets decreases.
- It tests in practice whether the database approach facilitates the parallelization of a piece of existing software.

Port the system to three types of hardware platforms

The implementation on a shared memory machine (SPARCcenter 2000) is already accomplished. The system will be ported to distributed memory computers (Meiko CS-2, and possibly IBM SP-2). If available in time, the virtual shared memory mode of the Meiko CS-2 will be exploited. In this mode, the local memories of the PEs form one contiguous address space.

The ports will be carried out to determine which are the important design aspects for each type of architecture to build a parallel in-memory database.

Test portability to other applications

The database system will be integrated with other applications, to test how promised features of the database approach work in practice.

9 Acknowledgments

We would like to thank Marcel Meesters, technical student at CERN, for his help in reviewing this paper. We would like to thank CPLEAR for providing access to the CPREAD program. This work was hosted by the ESPRIT GPMIMD project group at CERN.

10References

[1] Proceedings of the Eighth Conference in the series "Computing in high energy physics", Santa Fe, New Mexico, USA, April 9-13, 1990

[2] General purpose computing, *W.F. McColl*, Lectures on parallel computation, proc. 1991, ALCOM spring school on parallel computation, Cambridge, UK

[3] The Construction of a Small Communication Library, J.J. Lukkien, Computing science report, Eindhoven University of Technology, January 1995

[4] PVM 3 User's Guide and reference manual, *A. Geist et al.*, Oak Ridge National Laboratory, May 1993.

[5] Database system concepts, *Henry F. Korth and Abraham Silberschatz*, McGraw-Hill, 1991

[6] Zebra; an overview of the Zebra system, *CN, ECP and PPE divisions*, CERN, Geneva, Switzerland, February 1994

[7] The OPAL experiment at LEP, *OPAL collaboration (Janis McKenna, Dale Pitman)*, Phys. in Can.:50, 1994

[8] On-line event reconstruction using a parallel in-memory database, *Erco Argante*, GPMIMD deliverable, CERN, Geneva, Switzerland, April 1995

[9] Real-time database access on a massively parallel platform, *Erco Argante*, Thesis of the Post Graduate Programme Software Technology, December 1994

[10] ADAMO Entity-Relationship Programming System, Version 3.3, *Programming Techniques Group*, ECP division, CERN, October 1993

Real-time databases
An overview

M.P. Bodlaender
Department of Computer Science, TUE
The Netherlands

April 7, 1995

Contents

Chapter 1

An introduction to (real-time) distributed databases

Real-time distributed databases extend the power of centralised and distributed databases. Mechanisms are provided to incorporate the notion of time within the database semantics.Though databases already have a notion of time, since they try to compute as fast as possible, this is not sufficient in time-critical environments. In this chapter an overview of the differences between centralised, distributed and real-time databases is provided. Note that it is possible to construct a real-time centralised database as well as a real-time distributed database.

1.1 Centralised databases

The theory of **centralised databases** is well-developed, see for example [Pap79], [KR81], [YA88], [Vid85], [Vid91]. In general, arbitrary actions on a database have to satisfy the following two requirements: they must not disturb the logical consistency of the database and they must be efficient. The primitive actions that can be applied to the database are read and write actions. These actions access a single data item to either read or change its value. To be able to reason about database actions a **transaction** is defined as a collection of primitive actions (i.e. Read, Write) that is applied to the database.

Even when each transaction on the database leaves the database in a consistent state, a collection of transactions that is executed in an interleaved fashion can destroy that consistency. The (partial) order in which transactions are executed is called a **schedule**. If two transactions are unordered, their basic actions can be executed in any interleaved fashion. In articles [Pap79], [Vid85], [Vid91] correct schedulers are defined that order the transactions in such a way that database consistency is preserved.

The most important notion that has been developed is **serializability**: if a partially ordered schedule is serializable (proven equivalent to a totally ordered schedule), database consistency is ensured. Note that articles [GM83] and [GS85] illustrate that the class of serializable schedules is a strict subset of all consistency-preserving schedules.

Naturally, the transactions on a database must be efficient. Often, large amounts of data must be manipulated when complex transactions are performed on the database. The order in which certain basic steps are applied to the database has a great influence on the execution time of the transaction and a transaction manager that executes transactions in an efficient way is needed. In articles [IK94], [SY82] and [JK84] transaction management and query optimization are treated in depth.

1.2 Distributed databases

Recently, the wide-spread availability of computer networks calls for **distributed databases**. These databases try to exploit the properties of a computer network to increase the reliability, concurrency, capacity and speed of databases. A book that combines most aspects of distributed databases is [ÖzsuV91].

Why these enhancements can be expected from a distributed database is shown easily. Reliability can be increased because information can be replicated over multiple sites, thus lowering the probability that the crashing of a site leads to loss of information.

Because the database is actually divided into several smaller databases, it is often possible that small tasks are only performed at one or a few sites, leaving the other sites available for other tasks. This feature increases the amount of concurrency in the system, as multiple users can access the database at the same time.

In the current information age, large databases are needed to store all the information needed in complex organisations. However, the current state of hardware technology limits the size of a database a single computer can handle. The trend in computer architecture is towards a local area network of computers of intermediate size. These architectures are more powerful and are able to store unlimited amounts of data, as the size of the database can be expanded by adding an extra computer to the local area network. Therefore, mechanisms must be provided to deal with this fundamentally different architecture.

When a database is distributed over more than one computer, the computational power of the individual computers ceases to be the bottleneck of the architecture. While the maximal speed of a centralised database is dependent on technology of its CPU, this is not the case for distributed databases. This is because computations can be divided over the computers in the network. If a structural overload of the system occurs, it is possible to add more computation power to the system by adding extra computers to the network.

The bottleneck of the distributed database design is the communication cost.

If an information intensive transaction is processed that needs to access large parts of the network, the costs of communication rise rapidly. Even worse: the more computers participate in the distributed database, the more communication will be needed. Part of the research in distributed databases is directed at minimizing the communication cost of transactions that are performed on the distributed database.

1.3 Real-time databases

Databases have been used in various ways, but most applications of databases have been administrative. Databases typically try to fulfill two basic requirements:

- Operations on the database have to preserve the consistency of that database.
- The transaction throughput of the database should be as high as possible.

In real-time systems the computer interacts with an outside world that is constantly changing. Real-time systems often deal with temporal data, e.g. data that is only valid for a certain interval in time. This means that old data is as good as no data (i.e. Take data about the position of a moving object at some moment t. After several seconds the data will no longer reflect the position of the object in the real world. The data is no longer valid). Likewise, if a computer controlling a bridge decides that at time-interval $[t, t+d]$ it must be open because a ship will then pass, we don't want that bridge to be open long before time t or after time $t + d$, for this would hold the traffic longer than necessary. These two examples illustrate two extra conditions that we impose on **real-time databases** to preserve logical consistency:

- Internal data that represents the status of objects in the real world should accurately reflect the real status of the objects within an acceptable margin.
- Transactions of the database may only be executed in a certain time-interval. Most important, all transactions have a deadline after which the transaction fails.

In real-time databases schedulers should dispatch transactions such that they meet their deadlines. Therefore transactions that are nearing their deadline should be scheduled before other transactions. And in overtaxed systems that cannot meet all deadlines, we want to ensure that certain important transactions never fail, thus sacrificing other, less important transactions.

While in classical databases the primary goal is to preserve the database consistency, this is not always the case in real-time databases. For some applications it is more important that a transaction completes before its deadline than it is to preserve the database integrity. Therefore, current research is investigating the tradeoff between consistency and speed, see [KR92] or [KM93]. In a lot of applications, inconsistency can be tolerated as long as it is bounded.

1.4 Comparing the various database types

Each database-structure has been designed for a specific environment and with specific goals in mind. Low-cost centralised databases are very well suited for administrative purposes. The theory has been well-developed and 2PL (two-phase locking, a scheduling mechanism) is used all over the world.

Distributed databases offer all the services of a centralised database. More than a centralised database they offer concurrent access by multiple users. Data replication can make a distributed database more reliable than a centralised database. Distributed databases can easily be upgraded, as a good database design will allow for adding computers and storage to the distributed network.

Real-time databases explicitly deal with the notion of time. In applications where computers are used to control some environment they offer essential services. The most important service they provide is the meeting of transaction deadlines. A real-time database guarantees that, if the system is not overloaded, all transactions will finish execution before their deadline.

A priority mechanism can also be offered by Real-time databases. When the database cannot complete all transactions in time, it tries to ensure that transactions with higher priorities still meet their deadline. Thus real-time databases are also useful in areas where critical processes must be monitored along with less critical activities.

1.5 Organization of this paper

In the next nine chapters the main issues in real-time distributed database design will be briefly introduced. In no way an attempt is made to give a complete overview of the field, but hopefully the reader develops some global insight in the strengths and weaknesses of real-time distributed databases.

Chapter two is the justification of the research area, it provides a high level description of what services a real-time database offers and the resources that it needs to do so.

Chapters three to eight give introductions to different issues that relate to real-time distributed databases. In chapter nine it is observed that testing and comparison techniques used to date are fairly ad hoc and could use a more systematic approach. Chapter ten concludes with a summary of the issues that still need development in order to produce efficient real-time databases.

Chapter 2

What can a real-time database do for you?

The database design that has been used in many applications is a centralised, non-real time database. It provides access to the database to a limited number of users at the same time. Data-consistency is ensured and the database tries to execute as efficiently as possible. Distributed databases allow the databases to be implemented on a more general system architecture. They increase the reliability and availability of the database.

Real-time databases, centralised or distributed, deal explicitly with the notion of time. Data items in the database can reflect objects in the real world. These data items have to be updated by the real-time database, to maintain a correct view of the real-world. Also, the changing of a data item in the database may have effects in the real-world, for instance the movement of a robot-arm.

2.1 Real-time scheduling

To interact correctly with the environment, the real-time database allows transactions on a database to be scheduled according to some time based criterion. For each transaction t an interval $[s_t, d_t]$ can be specified such that the transaction t will not be executed before starting-time s_t, and t will be finished before deadline d_t.

If the information stored in the real-time database is used to derive an action that should be taken by the database somewhere in the future, it is possible to schedule this action. At the appropriate time it will be executed. This is best illustrated by an example. Suppose that inputs from an automated factory have been used to conclude that between 2am and 3am the workload is low enough to shut off the machines. With a real-time database it is possible to schedule two transactions, one at 2am and one at 3am that shut down and restart the machines, respectively. It can be seen that the real-time database can be used to interact with the environment, controlling parts of it.

It is important to realize that a number of different implementations of real-time systems are possible. These implementations could offer different services to the users, depending on the application of the real-time database. In the next sections some properties that a real-time system could provide are investigated. However, although these properties are often useful, they have their drawbacks. Therefore, not all real-time databases will offer all these properties. It should be clear that real-time databases must be tailored to suit each individual application.

2.2 Transaction priorities

In the ideal situation all transactions that are executed by the real-time database compute correctly and meet their deadlines. Unfortunately this is often not a very realistic assumption, the database can be confronted with an overload of transactions that all have to be completed within reasonable time. Even if the database is very efficient and fast, it could occur that it is unable to meet all deadlines.

In these situations, a number of transactions have to be cancelled. To provide the user with some control over the cancelling of transactions, each transaction is given a **priority** by the user. Now transactions with high priorities take precedence over transactions with low priorities if the database cannot meet all deadlines.

In general, this leads to an abort of an executing transaction, to allow high priority transactions to complete in time. The work that was already done by the aborted transaction is wasted. So a priority based scheduler degrades the throughput of the system. Although several schemes to reduce this degradation of throughput have been proposed, none of them do fully solve this problem. If throughput of the system is more important than the timely execution of individual transactions, user priorities should not be used.

2.3 Performance of real-time databases

In not-realtime databases the performance of the database is judged by its transaction throughput. This criterion is not satisfactory for real-time databases, as it does not take the deadlines of transactions into account. The performance of a real-time database is expressed in the number of transactions that meet their deadlines.

There is a sharp distinction between these two notions of performance. According to the real-time performance criteria, a database that processes thousand transactions in one hour, but misses each deadline by a few seconds is less efficient than a database that processes only hundred transactions which meet their deadlines. If the classic notion of database performance is used this would not be the case.

In the next two subsections two interesting techniques that can be used to increase the performance of the real-time database are mentioned.

2.3.1 Sacrificing correctness for performance

Whereas correctness is the main issue in classical databases, it is often more desirable to have some (partially) incorrect result on time than a correct result that arrives too late. Correctness can be traded for an increase in speed, raising the probability that transactions meet their deadlines. Of course, this is very application specific, but it is an interesting tradeoff that should not be forgotten.

A number of techniques have been proposed to bound the amount of inconsistency that can be allowed without invalidating the database to a point where it does no longer produce sensible output. For instance, it is not a big problem if a door-controlling computer opens the door once in a while without anyone present to enter the door. However, if it remains closed when people are waiting to enter, it is unacceptable. Another clear example is a climate controlling system. If it heats the room to 25 degrees, we find it irritating. But when the climate control decides that the room should be heated to 40 degrees, we shut it down as soon as possible!

2.3.2 Sacrificing generality for performance

A quite different approach that is used to increase the speed and throughput of real-time databases, is restricting the generality of the actions that can be applied to the database. As real-time databases are often applied for very specific purposes, this does not have to restrict the power of the database too much. If information about the types of transactions that will be processed by the database is available in advance, it is often possible to produce more efficient schedulers. This increases the performance of the database.

As a small example, suppose that it is known in advance that there is only one (periodic) transaction that writes to a data item. Other transactions only read the data item. With this information about the access behaviour of transactions, efficient scheduling of the transactions in question is possible. In fact, if a multi-version database is implemented, no concurrency control is needed at all! The writing and reading transactions can execute completely concurrent [1].

This technique can not be used in environments where no knowledge is available in advance or in environments where the transactions have no 'nice' properties that can be exploited for this purpose. Notwithstanding these negative observations, this can be a useful method to improve the database performance.

[1] For more information about multi-version databases see for instance [Wei87]

Chapter 3

Atomic transactions

One of the most important properties of database management systems is guaranteed data consistency. There can be various syntactic and semantic constraints on the information stored in the database. The technique that is generally used to enforce these constraints is the notion of **atomic transactions**.

3.1 Defining transactions

A **transaction** is a set of operations that is applied to the database in a certain order. Programmers of transactions have to ensure that the execution of a transaction on a consistent database leaves that database in a consistent state. Transactions are 'atomic', because either all the effects of a transaction are carried out, or the transaction doesn't take place at all. Other transactions will either see all effects of an atomic transaction, or no effects at all. In this way, the database consistency is preserved if all transactions are executed in a sequential way.

At some point, a transaction has to decide whether to complete the execution or to abort. This is called **committing** the transaction. Once a transaction has been committed, it is certain that all its effects are visible to other transactions.

3.2 Constructing transactions

It has been observed that information about transactions that is known in advance sometimes enables more efficient scheduling of transactions. Transactions are required to leave the database in a consistent state. In this section it is specified how transactions can be constructed.

3.2.1 Linear transaction model

The classic way to represent a transaction is as a list of read and write actions on data items. These actions are executed in a specified order. It is assumed that a transaction does some computation depending on the data items it reads and that some of the results of this computation are written back to data items in the database. Computations are not explicitly represented in this model. Typically, a **read action** or a **write action** accesses only a single data item in an atomic way (i.e. if the transaction fails, it does so between two basic actions, not during a basic action).

Linear transactions as a computation-model

The representation of transactions defined above is very well suited for reasoning about the scheduling of transactions and about interleaving of executions. This is because most scheduling is based on the "reads-from" relation. In general, transactions interact with each other by reading and writing data items. By representing a transaction as just a sequence of reads and writes the constraints of this interaction are explicitly captured.

The reads-from and writes-writes relations

The **reads-from relation** between transactions is defined as follows: a transaction t_1 reads from t_2 if t_1 reads a data item X whose actual value has been written by t_2. Analogously a **writes-writes** relation exists between t_1 and t_2 if t_1 overwrites the value of a data item X that has been written by t_2.

3.2.2 Nested transaction model

The transaction model presented in the previous section imposes only a simple structure on transactions. Worse, it supposes that each transaction is a sequential execution of basic actions. To express more general (concurrent) transactions while maintaining a strong grasp on the structure of transactions, the **nested transaction model** is introduced. Each transaction is represented as a hierarchy of transactions nested in transactions. Before, database consistency was required before and after the execution of a transaction. During the transaction, the database could be in an inconsistent state.

It is possible to require that each sub-transaction is a complete transaction itself: if it finds the database in a consistent state, it will leave the database in a consistent state. If this choice is made, the number of ways in which a transaction can be fragmented into sub-transactions is reduced. Thus it is harder to define transactions. On the other hand, it becomes possible to allow parts of a transaction to be used by the rest of the database while other parts are still in progress. Each sub-transaction can be regarded as a complete transaction. It can therefore commit without waiting for other transactions, as it leaves the database in a consistent state. A more fine-grained concurrency control is possible when sub-transactions are complete transactions themselves.

Nested transactions as a computation-model

In the nested transaction model a transaction is represented by a tree structure, where the leaves of the tree are the basic read and write events. This is a quite natural way to represent transactions. A lot of our programming languages are constructed as trees, where procedures are nodes and function-calls are links between nodes. Leaves are made up from the language-primitives.

If nodes that sequentially execute their children and nodes that execute their children in parallel are allowed, a very generic computation model is obtained. Some additional synchronisation between concurrent computations in different nodes can be obtained by communication between these computations. Allowing parallel execution within a transaction increases the amount of concurrency that the system allows, thus improving the performance of the system.

Use of nested transactions in a distributed network

Another benefit of nested transactions is that it is easy and natural to implement the distribution of transactions with them. The sub-transactions that have to execute on other sites than the initial transaction can be represented by sub-trees of the transaction-tree.

By representing the computation of each site by a sub-transaction, the execution of a transaction is defined by the execution of the sub-transactions and the communication between them. Communication between sites is often a bottleneck in distributed systems. By making the distinction between sites explicit in the model, it is possible to analyse the message complexities of transactions in terms of communication between sub-transactions.

Chapter 4

Concurrency control

Modern system designs have made it possible to execute processes concurrently, thus increasing the throughput of the systems. By concurrent execution of transactions the number of transactions that can be processed in a period of time is increased.

It is not possible to execute all transactions at the same time. A transaction that uses the result of another transaction has to wait until that result becomes available. Also, two transactions that both try to access a critical section (for example a printer) cannot run concurrently.

If two transactions are executed in parallel we imagine that their basic steps are executed in an interleaved fashion, not exactly at the same moments. This eases reasoning about concurrent transactions.

4.1 Concurrency and consistency

There is a strong relation between the amount of concurrency allowed by the databases and the maintenance of data-consistency. Transactions are designed in such a way that the execution of a single transaction leaves the database in a consistent state.

It is much harder to satisfy the consistency requirement if transactions are processed in an interleaved fashion. Other transactions can interfere with the execution of a single transaction, thus invalidating its execution. An example of this is given in figure 4.1. The consistency requirement is that accounts A and B sum to zero. However, due to the incorrect interleaving of basic actions of two transactions, this consistency is destroyed.

An explanation of the figure is probably helpful. Normally, a transaction is modelled as a sequence of read and write actions on data items. To show that arbitrary interleaving of transactions destroys the consistency of the database the internal computation of the transactions 1 and 2 is represented by the small statements. For each data item it accesses, a transaction has an internal vari-

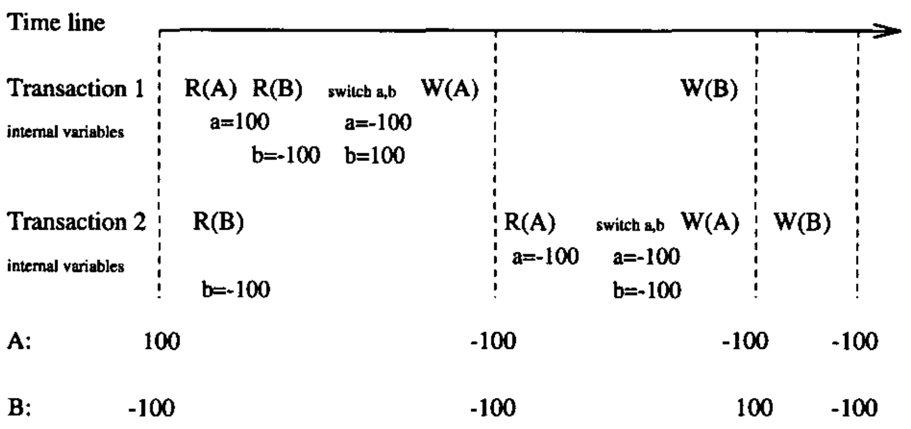


Figure 4.1: CONCURRENT TRANSACTIONS DESTROY CONSISTENCY

able representing that data item. If a transaction reads from the database, the result is stored in the corresponding internal variable. Four different "snapshots" show what the state of the database is. As would be expected, the database consistency is disturbed during the execution of the database. The database is still inconsistent after both transactions have finished execution. Therefore the schedule is incorrect.

The schedule show in 4.1 is incorrect because transaction 2 reads part of its data while transaction 1 is executing. During this execution the database consistency is not guaranteed, so transaction 2 can read from a (temporarily) inconsistent database. The correct execution of a transaction is only specified if a transaction reads a consistent database, no consistency requirements are placed on a transaction that reads from an inconsistent database. As can be seen from the example transaction 2 is capable of destroying the database consistency. Therefore a method to determine whether transactions can execute in parallel is needed. This is called **concurrency control**.

4.2 The serializability concept

A sequential execution of transactions always preserves the consistency of the database. This leads to the notion of **serializability**. A schedule s is called serializable if there exists some sequential schedule that has an equivalent effect on the database and executes the same transactions. In general the reads-from and writes-writes relationships of s should be preserved, and the final state of the database should be the same. This is called **conflict serializability**.

Theorem 4.1 *A serializable schedule is a consistency preserving schedule.*

Only an intuitive proof of the theorem is given. Two unrelated transactions can be executed concurrently or in a sequential way, without disturbing the database consistency. Consistency can only be broken by transactions that do have a reads-from or writes-writes relation. If these transactions are executed in an interleaved fashion the database consistency can be destroyed. Exactly this behaviour is prevented by the serializability requirement. Conflicting transactions are scheduled either before or after each other, but not interleaved.

To be able to maximize the amount of concurrency in the database, as many as possible schedules should be allowed.

Note however that there exist consistency-preserving schedules that are not serializable. Therefore, the set of serializable schedules is only a proper subset of the set of consistency preserving schedules. Checking that a schedule is serializable has been proven to be NP-complete.

A schedule is **legal** for a certain scheduler if it can be generated with that scheduler. Existing efficient schedulers all restrict the set of legal schedules to a subset of the serializable schedules, in order to reduce the complexity of generating legal schedules.

4.2.1 View serializability

It can be argued that the writing of a data item X that is never read before it is written again is useless. As no one has observed the writing of X, there would be no difference if the first write of X had never taken place. To represent this, the notion of view serializability is defined. A certain schedule is **view serializable** if it is equivalent to a sequential schedule that executes the same actions and preserves the reads-from relation between the transactions. Also the final states of the database should be the same. Note that the writes-writes relation between transactions is no longer important. Again, proving that a concurrent schedule is view serializable is NP-complete in the worst case.

It is interesting to note that if some writes are actually useless, the entire execution of these writes can be skipped. This assumes of course, that the overwriting transactions do no fail to complete their execution and abort.

4.2.2 Final-state serializability

Where view serializability abstracted from useless writes, final state serializability observes only the final state of a database. Intermediate states during the execution of a schedule are regarded as temporary states. Only the final result of the database is important. This assumption will probably not hold in real-time databases, where transactions can have a visible effect, not only on the database state but also on the real world.

A schedule is **final-state serializable** if it is equivalent to a sequential schedule that executes the same transactions. Equivalence of the schedules is now defined as equivalence of the final database states that result from executing the schedules.

4.2.3 Constructing workable schedules

Solving an NP-complete problem every time a set of transactions has to be scheduled is not a feasible option. Therefore, efficient schedulers that allow only a subset of the serializable schedules to be generated have been constructed. These schedulers can be regarded as heuristic methods to solve the NP-complete scheduling problem. Although they do not provide optimal concurrency, they introduce an acceptable overhead on the system.

A short description of the widely used **two phase locking** protocol is given and the difference between pessimistic and optimistic protocols is examined. Note that the two phase locking protocol serves as an implementation of the two phase locking scheduler.

Two Phase Locking

It is assumed that the scheduler is given a set of transactions T and a partial order $\prec$ on T. Transactions t_1 is ordered before t_2 if (wlog.[1]) t_2 reads or writes a data item X that has been previously written by t_1.

Suppose $t_1 \prec t_2$. The two phase locking protocol forces t_2 to wait until t_1 has finished, by locking data item X. A locked data item cannot be accessed by any other transaction, and t_1 does not release the lock until it is about to finish.

For simplicity it is assumed that only one transaction can have a lock on a data item, although optimizations can be made. So transaction t_2 has to wait or must abort, unless t_1 releases the lock on X.

Serializability is not yet enforced by this simple locking mechanism, but with a slight adaptation it will. Two phase locking (2PL) received its name from this adaptation: the protocol consists of a locking phase and an unlocking phase.

A transaction acquires all the locks it needs to execute in the **locking phase.** In the **unlocking phase**, a transaction releases its locks. Once the transaction is in the unlocking phase, it cannot obtain locks anymore.

The two phase locking protocol prevents the following, not-serializable behaviour: transaction t_1 locks X, writes X, releases X. Transaction t_2 locks, reads, writes and releases X. Transaction t_1 locks X again and reads it. This is not serializable: t_2 reads X from t_1. Therefore, t_2 must be executed after t_1. But t_1 reads X from t_2, so it should occur after t_2! This is a contradiction, so the scheduled transactions are not serializable.

The scheduler does not prevent deadlocks. Deadlocks may occur if two transactions need data items X and Y. Transaction t_1 has acquired a lock on X and

[1]without loss of generality

needs access to Y, transaction t_2 has acquired a lock on Y and needs access to X. Both must wait for the other transaction.

Optimistic versus pessimistic schedules

Two phase locking is a perfect example of a **pessimistic protocol.** It assumes that a lot of conflicts between transactions occur. Therefore, it does not execute a transaction until it is absolutely sure that it does not conflict with any other transaction in progress. This is ensured by the locking mechanism.

However, in a large database the chance that a conflict over a piece of information occurs between two transactions may be very low. If almost no conflicts occur transactions must unnecessarily wait for the locking of their own data items before they are allowed to execute. This observation has led to the construction of **optimistic schedulers.**

An optimistic scheduler first executes the transactions and then validates whether the transaction was executed according to a serializable schedule. If a conflict between two transactions occurs, one of them is aborted just before commit. It is still a point of study to determine under what conditions optimistic schedulers out-perform pessimistic schedulers.

Time stamps

Another well known method of scheduling uses **time-stamps.** Each transaction receives a unique time-stamp at some point. Now two transactions that both access the same data item have to be executed in an order that depends on the value of their time stamps. Time stamp schedulers can either be optimistic or pessimistic, depending on the moment that transactions receive their time-stamp and the moment that these time stamps are checked.

4.3 Weakening serializability

Determining if a schedule is a serializable schedule is an NP-complete problem. Efficient schedulers that produce serializable schedules never provide optimal concurrency because they use only heuristic solutions of the serializability problem. Also serializability does not completely capture the notion of consistency. To increase the amount of concurrency that schedulers allow, different approaches to database-consistency have been explored.

4.3.1 Epsilon Serializability

The first approach to mention is the notion of **epsilon serializability.** Epsilon serializability is a generalization of classic serializability. It explicitly allows some limited amount of inconsistency in transaction processing. This increases the concurrency allowed by the database as some not-serializable schedules are permitted. In particular, read-only transactions are allowed to run concurrently with update transactions. This might result in a inconsistent view of the database, but the database consistency is not affected. In general, Epsilon serializability bounds the amount of inconsistency that transactions are allowed to see.

Implementation outline

With each state of the database an amount of inconsistency is associated. This is defined as the distance of the state to a consistent state. Assume that a function distance(u, v) exists that defines the distance between every pair of states u and v. The database state space is **metric** if the distance function is symmetric and satisfies triangle inequality (for all states u, v, w holds: distance(u, v) + distance(v, w) $\geq$ distance(u, w)).

Now an epsilon-serializable schedule allows read-only transactions to run concurrently with update transactions if the amount of inconsistency they introduce is bounded by some import-limit. Likewise, an update transaction has some export-limit that specifies the maximum amount of inconsistency that it can export to concurrent, conflicting reading transactions. What limits can be allowed is dependent on the application that uses the real-time database.

Note that reducing the limits to zero gives us the classic serializable schedule. Pessimistic approximations of the amount of inconsistency can be computed if the database state space is metric.

4.3.2 Similarity Serializability

Similarity serializability is based on the observation that in real-time systems data items will never exactly match the status of objects they are describing. Similarity is a binary relation on the domain of a data object. Intuitively, two objects are similar if they are almost the same.

A schedule is view similar to another schedule if it schedules the same transactions and if these transactions read similar data. So intuitively, a transaction would in both schedules receive almost the same input. View similar schedules are only one version of schedules that are based on similarity. Whether two values are similar depends on the nature of the application of the real-time database.

Similarity of time

This concept is introduced to real-time database systems for the notion of time. Two measurements of an object that were taken at approximately the same time can be regarded as similar. This allows the use of slightly older values for read actions, even while the new values are being measured. This eases the problem of scheduling transactions in real-time. A discussion is presented in the chapter about time management.

4.4 Restricting transactions

Advance knowledge about the behaviour of transactions can enable us to do more efficient scheduling. All previous schedulers used the reads-from relation to govern the scheduling of conflicting transactions. However, if for example it is known that all data items are written by only one transaction, all transactions can execute concurrently if a version management scheme is implemented. In the next subsection the version management mechanism is explained.

By restricting the types of transactions allowed in the database the NP-complete serializability problem can be circumvented and efficiently produce highly concurrent schedules. This does of course limit the power of transactions. In the next section an example of a scheduler that exploits this property is provided.

4.4.1 Version management

A common transaction is the **read-only transaction**. Typically the user requires information and is not going to change the state of the database. The common occurrence of the read-only transaction justifies the separate treatment that is given here.

In a distributed database several transactions can be issued at roughly the same time. It is often not very clear in what order transactions should be processed. Therefore, if both a read-transaction and a write-transaction are issued and both transactions access the same data item, it does not matter in what order they are serialized. So for a read-only operation it makes no difference if it reads the most recent value or a slightly older one! Bearing this in mind, read-only transactions can be optimized by running them concurrently with update transactions.

Multi-version databases

To be able to serialize read-only transactions multiple versions of each data item are kept. Now if a read-only transaction is scheduled, it reads the latest, completed version of the data items it needs that are available at the moment the read-only transaction is scheduled. Update transactions can write newer versions of the data items that are being read, but this does not influence the outcome of the read-only transaction. With this construction it is always possible to serialize a read-only transaction. These transactions can always proceed with their execution.

Discarding versions

The existence of more than one version of each data item in the database places a huge demand on the resources that it can use. If old versions of data items are not discarded, the amount of data that needs to be stored by the database will grow out of bounds. A mechanism that discards versions that will not be necessary anymore is needed to make multi-version databases a viable option.

Depending on the exact scheduling mechanism a number of implementations is possible. A general solution is to keep the latest version always in the database and to keep track of the number of transactions that are still using an older version. If transactions are not allowed to be scheduled late (i.e. if a transaction arrives late it is aborted) old versions can be discarded as soon as no read-only transaction uses them anymore.

4.5 Handling deadlock and lifelock

Two important problems that should not be forgotten when designing schedulers are the problems of **deadlock** and **lifelock**. In a distributed database system these events cannot be locally detected. It is possible that a transaction t_1 waits on t_2 in site a, while t_2 waits on t_1 in site b. To be able to detect and do something about deadlock in a distributed system communication between the different sites of the database is needed.

4.5.1 Deadlock

If the database system makes use of some locking scheme to enforce serializable behaviour, deadlock may occur if two transactions lock a subset of the data items that they both need. No transaction acquires all its data items, so no transaction can proceed. They both wait on each other to release the locks they need. Deadlock can be prevented by checking that the "waits-for" dependencies introduced by the locking scheme are partially ordered. This means that no cyclic waiting may occur. If such a cycle exists, one of the transactions that is part of the deadlock has to be aborted. This checking is done by maintaining a so-called dependency graph. Vertices in the dependency graph denote transactions, and edges between vertices denote "waits-for" relations. An excellent overview of the theory of deadlock detection can be found in [Kna87].

Distributed deadlock

A deadlock in a distributed database can extend over more than one site. The information that is known about transactions at a single site is not sufficient to detect deadlocks. Several methods have been designed to detect deadlocks. A few methods are named without going into details.

- Transaction timeouts. If an upper bound of the transaction execution time is known, deadlock can be detected with the use of timers. If a transaction fails to terminate in time, a deadlock has occurred.

- Constructing a global dependency graph. If all sites send their local dependency graph to one site, all dependency graphs can be combined to produce a global global dependency graph.

- Chasing dependencies. If a site notices that a transaction it is processing is dependent on a transaction that is executing at another site, it sends the relevant dependency information to that site. If that site concludes that a cycle occurs (with aid of the received information), the deadlock is detected.

Methods to resolve the deadlock need to follow the detection. The methods are all based on the aborting of one or more transactions that are part of the deadlock. A problem in this area are "shadow deadlocks", i.e. the detection mechanism decides to abort transactions before deadlock has actually occurred.

4.5.2 Lifelock

Deadlock cannot occur in optimistic schedules, as transactions never wait. However, lifelock might occur. Lifelock is the situation that, although the database keeps processing transactions, a single transaction is never processed. Suppose that an executing transaction always finds out in the validation phase that it conflicted with a committed transaction. It has to abort the execution and reschedule.

Lifelock can be prevented if the scheduler can choose the transaction it aborts. In general, a transaction has to abort because it conflicts with a set of other transactions. If it is possible to abort this conflicting set, lifelock can be prevented by aborting the transactions that have aborted the least. This ensures that the oldest transaction in the system is not aborted. Eventually each transaction will be the oldest in the system or it will have committed. So eventually each commits.

Chapter 5

Reliability

Databases are meant to store information over long periods of time. With our current state of technology it is unrealistic to assume that the database system will never fail. Hardware errors, communication failures, software errors, almost anything can happen. It is possible to design hardware that uses redundancy to decrease the probability of a hardware failure. Likewise, software techniques are shown that prevent failures of the system to leave the system in an inconsistent state.

5.1 Failure models

Many different types of failures can occur, as was written in the introduction of this chapter. Two types of failures are recognised, based on the severity of the failure:

- **Fail-stop failures.** If a fail-stop failure occurs in a system, the system simply halts with its computation. After an unknown period, it restarts or continues its computation. When a system is able to continue its computation without losing its program state, the failure is called an **omission.** Omissions preserve the program state, but some results (messages) may have been lost. After a fail-stop failures, the program state has been lost and the system has to reboot. The time that a site needs to recover can be arbitrarily long.

- **Fail-insane failures.** When a fail-insane failure occurs in a system, the system doesn't stop, but it executes in an unpredictable way.

Fail-insane failures are more severe than fail-stop failures. The proof of this is simple: a fail-insane system can decide to behave like a fail-stop system. But it can also decide to continue the computation, acting quite normal but twisting its output. Conclusions based on this output will be incorrect. There is no fool-proof way of telling if a system behaves correctly, as the checking algorithm itself may produce incorrect output.

5.2 Maintaining consistency

In the previous chapter atomic transactions have been defined. The effects of an atomic transaction are either implemented entirely, or not at all. This property is used to maintain the consistency of the database. In this section methods to implement the behaviour of atomic transactions are examined. Transaction atomicity is preserved even when the system fails in the middle of a transaction.

5.2.1 Recovery from fail-stop failures

If the system fails in the middle of a transaction, this could lead to an inconsistent database. This happens for example when the system fails after half of the writes of a transaction have been carried out.

So the database has to be repaired when the system recovers. In the worst case, all main memory has been erased by the failure. Stable storage is needed to reconstruct the previous system state. **Stable storage** is a storage device (hard-disk, tape, etc.) that is failure-free. This is often implemented in hardware.

The undo/redo mechanism

In order to recover from failures all relevant transaction information is stored in a sequential file on stable storage. This file is called the **log**. Now before the results of a transaction are written, the previous values of the database are saved in stable storage. Then a "begin transaction" message is written to the (sequential) stable storage. Next, the updates of the transaction are actually carried out. When all updates have been applied to the database, "transaction finished" is written to stable storage.

The claim is that with this extra information, the atomicity of transactions can be ensured. Suppose the system has failed. Now when the database recovers, it reads its stable storage until it reaches the last "begin transaction" message. If an "end transaction" message follows, the system failed after completion of the transaction. Nothing needs to be corrected, the system is in a consistent state. If no "end transaction" message has been written to stable storage, the transaction was still in progress. All its writes are undone, by rewriting the previous state of the database that was saved on the stable storage.

The writing to permanent storage[1] may take place after a transaction has committed. If the system fails after the commit but before the actual write to

[1] With permanent storage the normal database storage (hard-disk) is meant. Note that the writing to stable storage is never delayed.

stable storage, it is impossible to undo the transaction. When this happens the log is used to redo the transaction.

This is the simple, centralised implementation of atomic transactions. Adaptations have to be made in a distributed environment but they will still be based on the existence of logs.

5.2.2 Handling fail-insane failures

Fail-insane failures are much harder to handle. The assumption of stable storage can not be made, as a fail-insane computer can overwrite its own storage. A solution could be a stable write-once, read-many storage. In this way, all correct actions of the system are preserved. It would be very hard to analyze this storage on recovery, because there will be no sharp boundary between the correct behaviour and the fail-insane behaviour of the system. I personally know of no results in this direction.

In a centralized system nothing can be done once a fail-insane failure occurs. One has to pray that it does not wipe out the entire database. Fail-insane failures can be handled to some extent in distributed databases. Replication of data prevents information to be destroyed by one fail-insane site.

5.2.3 Voting on actions

The adverse effects of fail-insane sites can be negated by voting on actions taken by the distributed database. An action on the database will only be executed by all sites if at least a majority of the sites concludes that it is a legal action. For these schemes to be successful, it is necessary that there is a bound on the number of sites that may fail-insane at the same time. Typically, at least a majority of the nodes participating in a vote must be correct.

5.2.4 Input certification

A noteworthy technique is that of **input certification**. An insane site that participates in a protocol does not need to send the same information to all sites. This can sometimes result in different conclusions in different correct sites. If the system tries to come to a global decision, this cannot be tolerated.

To prevent insane sites from sending different messages to different sites when they should be broadcasting a single message, a broadcast b from a site s that arrives at site t is not passed on to the controlling system. Rather, site t sends a message (s, b) to all other sites. This message effectively states "I received broadcast b from site s". Now if a node receives the same (s, b) message from at least half of the sites in the network, it accepts this message as a correct message. In this way, a fail-insane node can only send the same message to all nodes in the network.

5.3 Availability of the database

Another issue of dependability is the availability of the database. If a centralized database fails, the information stored is no longer available. But in a distributed network access to the remaining database sites in the network can still be provided.

5.3.1 Fault tolerance

By introducing redundancy in the database it is possible to make the system more fault tolerant. A very simple scheme that is used to build reliable computers is replicating the entire database X times. This X-redundant system can now handle $X - 1$ system crashes. If recovery mechanisms are provided, the system can handle $X - 1$ system crashes at roughly the same time.

5.3.2 Distributing data

There are several ways to store data in a distributed database. If the database is not redundant, each item is stored at a single site, the crashing of a site will prevent access to items stored at that site. In a lock-based system transactions that accessed data stored in the crashed site have to be aborted. Only transactions that use data items stored in surviving sites can continue execution.

Replicating data

Instead of the crude mechanism of replicating the entire database, single data items can be replicated and stored at more than one site. If one site fails, other sites are still able to provide access to all the information in the database.

There are several problems with this approach. Of course, replication of data reduces the overall capacity of the database. Algorithms that were simple and elegant in the not-replicated version become much more involved, if the system incorporates replicated data.

For instance assume that a transaction running at site t has locked a data item. Subsequently the site crashes. A mechanism has to be provided that releases all locks held by transactions on a crashed site. If no such mechanism exists the failure of a single site will prevent access to large parts of the database and no performance is gained from the replication of data.

So the design of the concurrency control mechanism should explicitly deal with the distribution of data. In the chapter on distributed systems, protocols that make use of replicated data to increase availability are discussed.

Chapter 6

Distributed systems

Distributed databases are useful because they enhance the reliability and availability of databases. They allow more concurrency than centralised systems and appeal to object-oriented programming approaches.

However, there is a price to be paid for these extra features. The database controlling protocols are more complex than in centralized databases and communication between sites is often a bottleneck. For instance, implementing a lock in a centralised database can be realised with simple semaphores. Implementing a lock in a distributed database requires the exchange of lock information between sites. If information about a lock is distributed over more than one site (to increase availability), the message cost grows in proportion.

In this chapter a few protocols are presented that are specially designed for distributed systems. This is meant to provide some insight in the complexities that arise in distributed systems.

6.1 Atomic commit protocols

One of the first problems that is unique for the distributed environment is the global commit. In a centralised database a transaction commits by writing a single message to stable storage. How this could be implemented in distributed databases is not instantly clear. A transaction consists of several sub-transactions. For each site that participates in the transaction a separate sub-transaction is defined. A protocol is needed to ensure that either all sub-transactions commit or that all sub-transactions abort. This is known as an **atomic commit.** All sites should agree on the same decision.

Decisions made by a site are un-reversible, and should be available within finite time. Finally, a transaction should commit if all of its sub-transactions commit, and no failure occurs. This property prevents the obvious solution of always aborting transactions.

6.1.1 Blocking

One additional feature that is important for the functionality of an atomic commit protocol is the so called **non-blocking property**. A protocol is blocking if the failure of a site that participates in the protocol blocks further execution. In particular: the protocol cannot abort and has to wait for recovery of the crashed site. The non-blocking property is not easily implemented. Theoretical results show that it cannot be guaranteed if no time-out mechanism or hardware detection of site failures exists. Therefore it is assumed in the rest of this chapter that such a failure-detection mechanism exists.

6.1.2 Two phase commit protocol

This is a simple, blocking protocol that offers just the basic services that we demand from an atomic commit protocol. It works as follows: The initiating site sends messages containing the necessary information for the sub-transactions to all sites. Once a site has finished its local computation it either aborts and sends "aborting" to the initiating site or it sends "ready". The initiating site receives all messages. If at least one message is an "aborting" message, the initiating sites sends "abort" to all participating sites and aborts. Otherwise it sends "commit". All participating sites receive the message and abort or commit accordingly.

6.1.3 Uncertainty of sub-transactions

Uncertainty is a fundamental property of (sub-)transactions. At the beginning of an execution the sub-transactions are not certain whether the transaction will commit or abort. The computation can still go both ways. At some point in the computation, the decision is made to either abort or commit by each site. Once it is possible that some site has decided on either of the two, no site may decide on an action without information about the decision in the other sites, for otherwise two different decisions could be taken.

With this property in mind, let us analyse the behaviour of the simple two phase commit protocol. At the beginning, no site is allowed to decide to commit. All sub-transactions can safely decide to abort. Therefore as soon as some site fails, the remaining sites abort the transaction.

The analysis becomes interesting once it becomes possible that some site has decided to commit. In the two phase commit protocol, the first site that decides to commit is the site where the transaction was initiated. Suppose some participating site p has sent its "ready" message to the initiating site and the initiating site fails before p has received the decision. Site p is now uncertain whether it has to abort or commit. Using some broadcast protocol it can try to gain certainty from other participating sites.

Suppose the initiating site is the sole failing site. The total set of messages that were sent to the initiating site can be gathered, so the decision that was taken by the initiating site can be deduced. If at least one other site failed, this does not apply. The remaining sites miss relevant information so they cannot infer what the initiating site was about to decide. When all remaining sites are uncertain, no site can decide whether to abort or to commit. The protocol is blocked.

It can be seen that the initiating site is never uncertain, so it can always decide on a course of action. This is because the initiating site is the first site that is allowed to decide to commit. Therefore, as long as the initiating site has not failed, the protocol is not blocked. Likewise, the protocol is not blocked if some remaining site has not yet sent its "ready" message or if some remaining site has already received the decision. The only scenario in which the two phase commit protocol becomes blocked is the scenario just described.

6.1.4 Non-blocking commit protocols

It is possible to construct commit protocols that have the non-blocking property. Instead of showing and analysing an entirely new protocol, it is briefly shown how improving a basic step of the two phase commit protocol does provide the non-blocking property.

Recall that the only scenario in which the standard two phase commit protocol is blocking, is when the initiator fails and at least one other site does the same. These two sites could have committed before they failed, so the remaining sites cannot abort. This is because they are uncertain about the decision of the initiator. Implementing an atomic broadcast suffices to realise the non-blocking property. An **atomic broadcast** is a broadcast where either all sites receive the message, or no site receives the message.

Achieving non-blocking with atomic broadcast

Now the initiator does not decide on commit until it has finished its atomic broadcast. If it crashes before it has broadcast the decision, it has not yet taken that decision, so the other sites can abort. If it crashes after the broadcast, all sites will have received the decision. Observe that in the previous protocol, delaying the decision till after the broadcast was not sufficient to provide the non-blocking property. This is because a participating site that received the "commit" message and subsequently failed could be the only site that committed if the initiator failed in the middle of the broadcast.

The implementation of an atomic broadcast is beyond the scope of this overview. It suffices to say that it can be achieved at an increased delay in time and with a higher message cost.

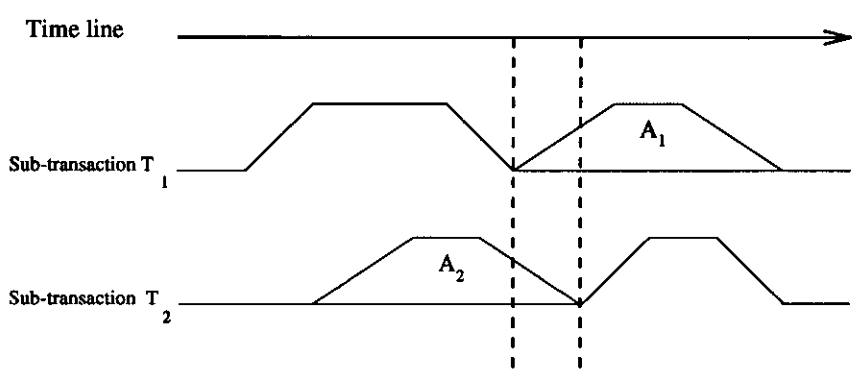


Figure 6.1: NOT-TWO PHASE LOCKING BEHAVIOUR

6.1.5 Global synchronisation

In many distributed algorithms a global synchronisation point is needed. An example of that is the commit protocol. The initiating site knows that all participating sites have progressed to a certain point (they have all sent their status messages), before it broadcasts its decision. So before sites decide to commit, all sites have at least responded once. Note that reasoning about time in distributed environment is a little more complex than presented here, as sites have no real notion of "global time".

Several different algorithms have been constructed that achieve global synchronisation. The algorithm described above is dependent on its initiating site. Other variants have been designed that increase robustness, decrease time complexity or decrease message complexity.

Distributed two phase locking

To be able to design a distributed version of the two phase locking protocol a global synchronisation protocol is needed. Recall that essential for the two phase locking protocol was the existence of a locking phase and an unlocking phase.

Suppose the two phase locking protocol is used to schedule a distributed transaction. It is not sufficient to ensure a local two phase behaviour, as the sites are not synchronised in time. In picture 6.1 an example is given of not-two phase locking behaviour that arises because the sites are not synchronised. Sub-transactions T_1 and T_2 execute on different sites that are not synchronised. Because of communication delays or because of the difference in speed of the two sites the sub-transactions do no start and stop their locking and unlocking phase at the same moment. The phases are so far apart that T_2 begins its locking phase after T_1 has finished its unlocking phase. Another transaction A could now read the results of T_1 and write the data items that T_2 is going to use. This not-serializable behaviour exists because transactions have no global synchronisation point between the locking and unlocking phase.

6.2 Availability of data

Data that is stored in the database should be accessible at all times. Even if some of the sites fail, one would like to manipulate data. This is clearly impossible if data is stored at a single site. However, if data is replicated over multiple sites, it will be available as long as at least one of the sites remains functional.

Although data replication increases the availability of data, it introduces problems for maintaining correctness. Recovery management is needed to update sites that recover from crashes, as changes will have been made to data items that are also stored at the recovering sites. But most important, the concurrency control algorithms have to deal with the replication of data.

Access to a data item is no longer centralised at a single site, but is distributed over the network. Producing correctness preserving schedules requires communication between the sites in the network. A number of distributed concurrency control algorithms are mentioned.

A simple strategy

Each copy of a data item is treated as a separate data item. If a transaction wants to read or write an item it has to obtain locks on all copies. Obviously this leads to a high communication and storage cost without an increase in concurrency or availability. To design an efficient concurrency control algorithm, mechanisms are needed that increase concurrency and provide access to data even if a few sites fail.

Single read-lock strategy

A minor adaptation to the previous scheme is that a transaction that just reads a data item X only locks the local copy of X. In this way, read actions can be executed concurrently. Write actions still conflict with other writes and reads. Read actions are not blocked if a site fails, write actions have to lock all copies and cannot proceed.

Primary copy strategy

This section is concluded with presenting a simple version of the primary copy protocol. This protocol maintains a high level of availability, even if some sites fail.

For each data item a **primary site** is defined. The copy of the data item stored there is the primary copy. All other copies are backup copies. Transactions request read and write locks only at the primary site. Therefore, actions are not blocked as long as the primary site remains functional. Other adaptations of the primary site protocol deal with the crashing of the primary site. Still, the simple primary copy strategy is an improvement over the single read-lock strategy where an arbitrary site failure would block the protocol.

6.2.1 Network Partitioning

If the network that is the foundation for a distributed database becomes partitioned it would be nice if the two separated parts of the database would remain functional. Information that has only been stored in one partition is unavailable for the other partitions of the database. Transactions that act on this information can only execute if they are issued in the same partition.

So network partitioning cripples the performance of the database in the unreplicated case. But problems are not over if the available data in the database is replicated. If update transactions are applied to data while the network is partitioned, it is possible that two different values are assigned to copies of the same data item. If the network is connected again the database is no longer consistent. Only read-only transactions are allowed in all partitions while the network is partitioned. Update transactions can be allowed in one partition. If updates would be allowed in more than one partition, two or more different copies of one data item can exist. If updates are only allowed in one partition, all other partitions can use the old copy of the data item. (As the network is partitioned, it is certain that transactions running in different partitions can be serialised by putting all the transactions from the read-only partitions before the update-partition).

Chapter 7

Time management

In conventional databases information is static: as long as no transaction changes the information, it does not change. This is not the case in real-time databases. Often, information loses value as it grows older. This is especially the case when the information in question is a representation of the real world (hence the name real-time databases), as the real world changes in time. Likewise, if two pieces of information are gathered at completely different times, they do not relate to each other.

7.1 Temporal consistency

From the observations just made **temporal consistency** can be formulated in two components:

- **Absolute consistency**. A direct relation must exist between the state of the environment and its representation in the database. If the system has an incorrect view of the environment its actions will be nonsensical.

- **Relative consistency**. Data derived from the environment must be temporally consistent with the other data that has been derived.

Examples of both absolute and relative temporal consistency are given. Suppose a computer is used to monitor the amount of people in a room. If it counts the number of people every five seconds, it will always have a good approximation of the number of people in the room. If it counts once every hour and everybody leaves after thirty minutes, the representation in the computer would no longer reflect the real world, it would be inconsistent.

Now suppose the computer monitors two rooms. First it counts all people in room one in five seconds, then all people in room two. However, while the computer was counting, people were switching rooms. If both counts were ten, can it be concluded there were twenty different people in the rooms? If the rooms are adjacent about five people could have switched, so only fifteen different people are needed to arrive at our result. The two counts would not be temporally consistent.

But when the rooms are fifty meters apart, almost no one could have crossed that distance in five seconds, so there are indeed twenty different people. The results of the two counts are then indeed temporally consistent, i.e. the fact that in the real world people cannot be in two rooms at the same time combined with the data allows the computer to conclude that there are at least twenty different people present.

7.1.1 Absolute consistency

Information that reflects the real world is only valid for a certain interval in time. The length of this interval is dependent on the nature of the object that is represented and the amount of inconsistency that is allowed by the system.

If an object is changing rapidly, the interval during which information is valid will be short. If the database requires that information about an object may only deviate five percent from the real status of the object, the information may have to be refreshed more often than when it is allowed to deviate ten percent.

So far, it has implicitly been assumed that the behaviour of objects is predictable. If an object can suddenly change its entire state, it is impossible to prevent inconsistencies to exist in the database. Likewise, linear behaviour of the information about objects has been assumed. When a small change in data reflects a major change in the state of the object, even small inconsistencies in data can lead to totally wrong conclusions. For example, if a five degree error is allowed in the temperature of water, the difference between ice and water if it is just below freezing point cannot be specified. While for normal temperatures a deviation of five degrees might be acceptable, it is not acceptable around the freezing point of water.

If it is impossible to refresh the information stored in the database often enough to maintain an acceptable representation of the real world, one can use the predictable behaviour of objects to extrapolate the history of an object. If bounds on the speed with which an object can change its status are known, numerical methods can be used to bound the error that is made in the prediction.

7.1.2 Relative consistency

Related data about objects in the real world is only consistent with each other if the data was gathered at approximately the same time. This is called the relative consistency of data. As with absolute consistency, relative consistency is dependent on the speed with which the represented objects are changing and the amount of error that is permitted.

But where absolute consistency is only preserved for a small interval in time, relative consistency is a permanent property of a pair of data items. Data items are consistent if they have been gathered within a specific period of time from each other.

Relative temporal consistency is not a transitive relation, if A and B are temporally consistent, and B and C are temporally consistent, it is not necessary that A and C are temporally consistent.

This is easily illustrated. Suppose data items are temporally consistent if their age does not differ by more than five seconds. Now A is gathered at time 10. B has been gathered at time 14. Clearly, A and B differ only 4 seconds and they are temporally consistent with each other. Now C has been gathered at time 17. B and C differ only 3 seconds, so they are consistent. But A and C differ 7 seconds, and they are not relatively consistent.

7.2 Time critical scheduling

In a real time environment transactions will have a time-interval associated with them. The real-time database must ensure that each transaction is executed within its own time-interval.

7.2.1 Time based scheduling

The problem of scheduling a set of transactions that all have time-intervals in which they have to be processed on a database with limited computation power is NP-complete. This does not even take into account that the transactions also have to be executed in such a way that the database consistency is preserved. Finding a serializable schedule, the most common notion of database consistency, is in itself an NP-complete problem. It is therefore unrealistic to assume that the optimal solution to the time based scheduling problem can be found.

Behaviour under overload

If too many transactions have to be processed in a time interval, there will be no solution of the scheduling problem. The database is unable to process all transactions in time. We would like a scheduler that even under these circumstances processes as many transactions within their intervals as possible.

Transaction execution length

To do any intelligent scheduling, information about the execution length of transactions is needed. If no such knowledge is present the optimal strategy is to schedule transactions as early as possible. The notion of **slack time** is important. The amount of slack time that a transaction has is the length of the interval in which it is allowed to execute minus the amount of time that it needs to execute. A correct schedule is easier to find if transactions have a lot of slack time.

7.2.2 Missing deadlines

In an overloaded system, the scheduler will not be able to execute all transactions within their associated time-intervals. Where in normal databases it is possible (though not desirable) to queue incoming transaction until workload decreases and the system catches up, in real-time databases the value of a transaction will dwindle away once its deadline has been missed. Dependent on the type of transaction three types of deadlines are recognised: soft, firm and hard deadlines.

Soft deadlines

Transactions with **soft deadlines** do not lose their complete value once they have passed the deadline. Instead, the value of a transaction that has passed its soft deadline slowly dwindles away. The most famous example of this is the large project (be it bridge-building, software construction, whatever). The number of times that projects do not make their deadlines is staggering. However, most of them are still completed. It is often better to have finished a project too late, than not to have finished it at all. Clearly, a project cannot go on forever. If no goal is within sight, eventually funding will be stopped, the project will be cancelled, its value is decreased to zero.

Firm deadlines

Transactions with **firm deadlines** do lose all their value once the deadline has passed. There is no use in continuing the transaction. Examples of this are all around us. Think of going to the supermarket. If you are half-way to the supermarket and it is closing time, there is absolutely no sense in continuing your trip.

Hard deadlines

The last type of deadline that is recognised is the **hard deadline**. If a database fails to execute a transaction with a hard deadline in time, not only does the transaction lose all value, but this failure imposes a heavy negative value on the system. Examples of this lie for instance in computer-controlled security systems. A crude example is the computer monitoring a nuclear power plant. Once the reactor temperature rises above a certain limit, the computer must activate the emergency cooling system. If the computer fails to react in time, a major disaster might occur.

7.3 Priority scheduling

The existence of different types of transactions that each have a different impact on the system, should their deadline be missed, leads to the introduction of **transaction priorities.** Clearly the priority of the emergency cooling system is higher than the priority of the daily memo delivery program. The mechanism of priorities can be provided as a service to the users of the real-time database or it can be an implementation of the different types of deadlines that transactions have. On the other hand, priorities can also be used as part of the implementation of a scheduling protocol. For instance in the "Earliest Deadline First" protocol, transaction priorities are defined as the inverse of their deadlines.

7.3.1 Defining priorities

Before priorities for all transactions are defined, it must be defined what these priorities exactly mean. In most literature, if transaction t_1 has a higher priority than t_2, t_1 will always have precedence over t_1. It is also possible that the scheduler tries to maximize the total sum of priorities of transactions that are executed. That would mean that a transaction of priority five could be aborted by five transactions of priority one.

Imagine a vending machine. Its goal is to earn as much money as possible. Now if it spends too much time with a customer that is about to buy a very expensive article, it can better spend that time selling cheap articles to multiple customers.

The normal priority scheme, where transactions of higher priority always take precedence is more suited to implement critical processes, so in the remainder of this chapter such a priority scheme is used.

7.3.2 Handling priorities

Suppose that in a very general scheduler a high priority transaction has a conflict over a data item with a low priority transaction. One transaction has to wait or abort. Suppose too that the low priority transaction is already being executed. Two options are clear: aborting the low priority transaction, or letting the high priority transaction wait.

Aborting low priority transactions

A transaction with a high priority takes precedence over transactions with low priorities. A simple implementation to enforce this rule is to abort all low priority transactions that conflict with a high priority transaction. This mechanism ensures that the transaction with the highest priority will always be allowed to execute. There are several drawbacks to this scheme. First of all, even transactions that were very near commit point are aborted. This results in a large waste of database resources and reduces the overall throughput of the system. Several schemes have been designed to remedy this problem to some extent. In general, they abort transactions that are in early stages of their computation and allow transactions to finish if they are nearly done.

The second problem is that low priority transactions can be life-locked by this mechanism, if there are a lot of high priority transactions. Often if a transaction with a low priority is aborted several times, its priority will rise. Take for example maintenance: missing one maintenance checkup is not very important. However, regular maintenance is essential for a system to keep functioning in a reliable way. Maintenance cannot be postponed indefinitely. This results in a "race for priority" that can disrupt the entire priority scheme.

Priority inversion

If a high priority transaction that is not yet nearing its deadline has a conflict with a low priority transaction, is does not need to abort that transaction.

Instead of aborting the low priority transaction, it runs to conclusion. But as the high priority transaction is now waiting on a low-priority transaction, it is effectively blocked as the low priority transaction will have to wait on higher priority transactions.

To remedy this problem it has been suggested that the low priority transaction inherits the high priority of transactions that are waiting for it to finish. However, this means that a (once) low priority transaction is now allowed to abort medium priority transactions. This is called the problem of **priority inversion**.

Chapter 8

Integrating operating system & database design

Normally when researchers start investigating a subject, an abstract representation of a real problem is formulated. The research focuses on one aspect, instead of looking at the big picture. Once a solution to such an abstract problem is found, this solution is translated back to the real environment and implemented.

Most of the time it is assumed that real-time databases are built on some operating system that offers storage services. The properties of these operating systems are only roughly defined. If the analysis of the database is combined with the analysis of the operating system, more realistic assumptions about the reaction time of the operating system can be made. This enhances the time-control and the precision in which the length of transactions can be predicted.

Operating systems offer file storage services, much in the same way that databases offer more fine-grained storage services. By combining the database with the operating system this replication of services can be avoided, thus reducing the overhead imposed by the system.

These two observations justify the combining of database design and operating system design. In this chapter it will be investigated what can be gained from this combination.

8.1 Data caching

To increase the efficiency of the hard-disk it is useful to keep some of the information on the hard-disk stored in main memory. Even if the same information is accessed multiple times, only two disk accesses are needed. One to get the information from the hard disk and one to update the hard disk before the main memory is erased. This technique is called **data caching.**

In large database systems, the cache cannot hold all information that is retrieved from the hard-disk. At some moment, information stored needs to be erased to make room for new data items that are not yet in the cache. The efficiency of the caching-mechanism depends on the selection of data items that are removed from main memory and stored back to disk.

It is important to take the disk cache explicitly into account during real-time database design. There are two major drawbacks that have to be considered:

- System crashes. If at some moment the system crashes and the main memory is wiped, all changes to data items that were cached are lost. If the disk is cached, it is not certain that a write to the disk is instantly carried out. The recovery mechanism has to be adapted to cope with this extra complication. In general all transactions will be redone whose results might have been lost in the crash.

- Transaction time bounds. In a real-time system tight bounds on the execution time of transactions are necessary to do intelligent scheduling of transactions. If the behaviour of the caching mechanism is not analysed only the worst case scenario can be assumed: the cache is full, data has to be written back to the disk before the new data is retrieved and stored in the cache. So although caching does increase the performance of the hard-disk, it degrades the worst-case analysis as for a single read operation at least two disk-accesses will be needed instead of one.

8.2 Virtual memory

Another technique that is frequently used in operating system design is virtual memory. The actual main memory of a computer is often not large enough to completely hold very large programs. The CPU can only access a very small portion of that memory at the same time (typically one or two locations). Therefore large parts of the main memory will not be accessed for some time.

The technique called virtual memory makes use of this property by allowing programs to use more main memory than what is actually available. If a program accesses memory that does not exist, the (virtual) memory manager stores a currently unused part of main memory on hard-disk and offers the now free memory to the program. If main memory that is stored on hard disk is accessed by a program, the memory is retrieved and some other part of main memory is swapped to the hard-disk.

The virtual memory mechanism actually uses the hard disk as slow main memory. To make optimal use of the available (fast) main memory several swapping algorithms are possible. Nevertheless, virtual memory degrades the speed of main memory access.

A tradeoff between available memory and memory speed might be envisaged, if the degradation of speed would be a gradual process. But this is not the case. Memory access is a fast as normal until a program (or transaction) accesses memory that is stored on hard disk. That memory access initiates the swapping of memory to and from the hard disk. This is disastrous for a worst case analysis of the execution time of transactions.

8.3 Conclusion

As illustrated by the examples above, a lot of practical problems surface if the solutions to database problems are exported from an experimental environment to a real environment. Especially, the worst-case execution time of a transaction is affected by disk-IO. Without previous knowledge about transactions, cache hits and page swapping cannot be predicted. As in real-time databases timeliness is often more desirable than fault tolerance, a main-memory system with delayed writes to disk may be more effective.

Chapter 9

Analysis of database designs

In real-time database management systems, it is not so important that the database has a high transaction throughput, but rather that each individual transaction has a high chance of completing before its deadline. Although these two notions overlap, they are not the same, as has been illustrated in an earlier chapter.

9.1 Existing results

The analysis of the efficiency of real-time database designs has been rather rudimentary. Almost all articles that deal with efficiency give simulation results. Although simulations can be very useful for comparisons between schedulers they lack the thoroughness of the analytical approach. Relative few articles have been written that analyse not-realtime database efficiency instead of using simulations.

In article [YDL93] both two phase locking and pure optimistic concurrency control are analysed using Poisson processes. This paper presented an analytical approximation of the average transaction length, given that the transactions arrive at the scheduler as a Poisson process. Unfortunately the analysis of the two schedulers is mixed, which muddles the article. This distracts from some important assumptions that where made to arrive at the result.

To be able to say anything about the probability that a transaction will finish before its deadline, the average execution time is insufficient. The analysis of the complete probability distribution instead of the average execution time is in general much more involved.

9.2 Comparison problems

An analysis of a scheduler should result in a set of success probabilities for an arbitrary transaction under a given workload. This probability will depend also on the transaction length and the size of the database. Even if a distribution can be specified, it is not easy to compare the efficiency of different real-time schedulers. This is because the efficiency of a scheduler depends on a number of parameters:

1. **Centralised versus distributed environment.** The communication delay introduces problems that are very specific for distributed systems. At the same time, distributed systems offer more computation power. It is clear that distributed systems are unsurpassed if availability is the criterion.

2. **Read-only queries versus updates.** More efficient scheduling is possible if read-only queries are treated as a special type of transaction. Dependent on the application of the database (mostly the percentage of transactions that are read-only transactions) this optimization will be more or less useful.

3. **Real-time scheduling versus normal scheduling.** The performance criteria for real-time databases and normal databases differ. A comparison between a real-time database and a normal database is therefore complex, but it might be useful to analyse the behaviour of a real-time schedule in a non-realtime environment and vice versa.

4. **Priority based or not.** A real-time scheduler can offer a priority mechanism to the users, to give them some influence over the behaviour of the scheduler under a high system load. Of course, this affects the efficiency of the scheduler.

5. **Conflict-rate.** Schedulers behave differently under different system loads. A scheduler can be very efficient under a low system load, but lose performance as soon as the system load increases. Another scheduler can have a rather constant performance, not perfect under relatively low system loads, but handling well under high loads.

6. **Amount of possible deadlocks.** Some schedulers do not prevent the possibility of deadlock. While this permits them to execute more efficiently, deadlocks have to be detected and resolved. Dependent on the nature of the transaction system, deadlocks can be allowed to exist for some time before they are resolved. Especially in a distributed system this reduces the message cost that is associated with deadlock detection.

7. **Variance of transaction lengths.** Several schedulers perform well as long as all transactions are of the same execution length, while degrading when lengths of transactions vary. For instance, pure optimistic concurrency control can lifelock long transactions if a steady stream of short transactions enters the system. Two phase locking would not suffer from this problem.

8. **The level of reliability and availability that is required.** Reliability and availability of the database are desirable properties, but they come at a cost. A distributed scheduler that implements no global commit protocol is unreliable in case of site failures, but is very efficient. A distributed scheduler that uses the two phase commit is less efficient but reliable, but suffers from blocking, thus reducing availability. A scheduler that uses the three phase commit protocol is both reliable and does not suffer from blocking, but the three phase commit protocol introduces more overhead than the other two approaches.

9. **Required advance knowledge about transactions.** Schedulers that rely on access invariance[1] can overcome the problems of lifelock and deadlock easily. Consequently a more constant response time can be guaranteed. The rate-monotonic scheduler is a perfect example. This scheduler knows that all transactions are periodic, with deadlines equal to the beginning of the next period. All transactions can be preempted and continued later. The last assumption distinguishes the allowed transactions of the rate monotonic scheduler from transactions that are normally allowed by databases.

9.3 Conclusion

Very few articles deal with the efficiency of transaction schedulers in an analytical way. In the field of real-time systems, where transactions lose their value once their deadline has passed, guarantees about transaction execution times are even more important than in normal databases.

In normal databases, the throughput of the system is of primary concern. Such throughput can be easily measured in a testing environment. In real-time systems, the execution time of each individual transaction is important and more elaborate testing techniques are necessary. At the same time, the analysis of the transaction execution time becomes more involved, as the average execution time no longer suffices.

It will probably be hard to give sharp analytical results, as the problem has remained almost without results for so many years. Research should start with analysing simple schedulers with certain restrictions on the transaction types and frequencies. Nevertheless, the field of real-time schedulers does need a fundamental basis, that cannot be completely provided by test-results.

[1]The data items needed by a transaction are known in advance.

Chapter 10

Research issues

After presenting this overview of the field of real-time distributed databases it is time for some reflection. Although a lot of good results are already available in the separate subfields it is not instantly clear that we are now able to build the optimal real-time database with these mechanisms. In this chapter the overview is completed by pointing out the areas where further research is still needed. This will be contrasted by a short summary of results that are already known.

10.1 From user-interface to implementation

The real-time distributed database stores information that corresponds to the real world and offers various services to its users. Instead of restricting the notion of users to humans, users can range from other computers to air passing a pressure valve. This wide range of users has no knowledge of the database structure and high level services have to be provided.

These services are typically provided by transactions that have been programmed in advance. Transaction programmers deal with input/output devices and prefer to represent actions in an abstract language, independent from the underlying implementation of real-time databases. Algebraic or relational languages exist that allow arbitrary complex database transactions (for example SQL). Classic languages do not deal with real-time aspects and languages that do take real-time into account are just beginning to emerge.

The translation from algebraic (or relational) operators on an abstract representation of the database to actual distributed transactions is the domain of transaction managers. Different translations of an algebraic expression can differ exponentially in execution time and message costs of the resulting transactions (in a distributed environment). Finding the optimal transaction corresponding to an algebraic expression is NP-hard. Several heuristic Transaction Managers have been developed that offer good approximations.

10.2 Transaction scheduling & correctness

To increase the efficiency of transactions it is beneficiary to execute transactions in parallel. However, the database consistency is defined only between transactions. During a transaction, a temporary database inconsistency is allowed to enable efficient execution. If transactions are allowed to execute concurrently, transactions might read data that are temporally inconsistent and act as if the data are consistent. Permanent inconsistencies can occur. To determine whether two transactions can execute concurrently the database system provides a transaction scheduler.

The transaction scheduler tries to maximize the amount of concurrency (executing transactions in parallel) while it preserves the database consistency. Again, finding the optimal schedule is NP-complete. Existing transaction schedulers preserve consistency at the cost of reduced concurrency. As these schedulers are simple approximations of the optimal schedule, they can be improved.

10.2.1 Transaction classes

It has been observed that for several classes of transactions the scheduling problem is not NP-complete at all. A taxonomy of transaction classes that have nice properties that allow the scheduler to generate optimal schedules efficiently can be very useful but does not (completely) exist. A well-known class consists of read-only transactions. All read-only transactions can execute concurrently.

10.2.2 Periodic transactions

A lot of scheduling research has gone into the the scheduling of transactions with hard-realtime constraints. These transactions are not allowed to fail, they have to run to completion within their execution interval or otherwise the entire schedule is incorrect. A lot of these schedules were constructed at pre-runtime. Therefore the complete set of transactions that was to be scheduled was known in advance. Often the scheduled transactions are periodic, i.e. a transaction is at regular intervals or sporadic, i.e. a transaction is run at regular intervals, but may skip some of these runs.

The periodic nature of transactions can probably be exploited as well in soft real-time scheduling systems. In soft real-time, the number of successful transactions is optimized. It is permitted that some transactions fail to meet their deadlines, as long as it is a small percentage of the total number of transactions. An optimal soft-realtime schedule can always be found, opposed to a hard real-time schedule that may not exist. An optimal real-time schedule of periodic transactions should probably guard against life-lock.

10.2.3 Allowing inconsistencies

Other research tries to increase the amount of concurrency allowed at the cost of introducing inconsistency in the database. Such a scheduler can be useful if the amount of inconsistency introduced is somehow bounded. Especially in real-time databases it cannot be avoided that inconsistency in data gathered from the real world occurs. Therefore this seems a natural way to increase concurrency.

10.3 Real-time transaction scheduling

In a real-time database, transactions are only allowed to execute within certain time intervals. A scheduler that not only preserves data consistency but also ensures that all transactions are executed in their interval has to be provided. The problem of generating a schedule that executes all transactions within their intervals in an environment with limited resources is NP-complete.

Ordinary schedulers can be slightly modified to incorporate deadlines. Unfortunately most schedulers behave badly under high system loads. An ordinary scheduler will try to execute all transactions. If the system load is high, this will mean that all transactions run to completion, but also that almost all transactions will have missed their deadlines. A real-time scheduler must decide to abort transactions that miss their deadlines in order to complete a (constant) number of transactions in time.

A lot of research has been directed at hard real-time scheduling. In hard real-time, not even a single transaction is allowed to fail. This is a very restrictive constraint, that often cannot be realised. Also a common assumption is that transactions can be pre-empted, that is put on hold and resumed later. This is quite contrary to the correctness constraints of databases, where the database consistency is temporarily disturbed during a transaction. It therefore seems more logical to use the existing correctness preserving algorithms as a starting point instead of using the real-time scheduling algorithms as a starting point for real-time database schedulers.

10.3.1 How many resources are required?

Surprisingly little analytical studies have been published about transaction execution times, and I know no analytical results in real-time scheduling. The performance of a soft-realtime transaction scheduling mechanism can be defined by the probability that a transaction executes within its time interval. Several simulation studies have been made to determine these probabilities, but no analytical analysis of transaction schedulers are available. An analytical analysis would present us with a set of hardware requirements, as well as clear assumptions on the behaviour of the users of the real-time database.

Associated with this probability it is interesting to specify the relation between the performance of the real-time database and the hardware that supports the database. If for a given real-time scheduler this relation can be specified an estimate of required resources can be given analytically for a given problem.

10.3.2 Disk-based systems

As mentioned in the previous chapter, a lot of timing problems arise when the underlying operating system optimizes disk access by buffering, or when virtual memory is implemented. Periodic cache-flushes could be scheduled when no transactions are in progress. This would prevent unreliable reaction times of the operating system, at the cost of an extra transaction (the flushing of the cache) that has to be scheduled.

10.3.3 Combining correctness and timeliness

Scheduling transactions in such a way that consistency of the database is preserved while offering optimal concurrency and scheduling transactions within their execution intervals are related. As these problems are tough to solve on their own, they are often treated separately. In real-time databases, a scheduler has to be provided that takes both requirements, correctness and timeliness into account. The current approach is to use existing, correctness preserving schedulers and prove that under a restricted workload sufficient transactions meet their deadlines.

10.4 Distributed transactions

The distribution of a database can increase the availability, reliability and capacity of the database. This does come at a cost. First of all, communication between the different sites of the database becomes an important factor of time-delay. Secondly, scheduling of distributed transactions becomes more involved because of distributed deadlocks, global correctness and routing problems.

10.4.1 Communication delay

As mentioned, communication delay is an important factor in distributed databases. Therefore algorithms that were fairly trivial in a centralised database have to be optimized in the distributed environment. A way to reduce message costs is to replicate data over the different sites, but this introduces new consistency problems. Several solutions have been proposed and exist, but the field is still under development.

10.4.2 Query optimization

The transaction manager that optimizes transactions to reduce the number of execution steps of a transaction has to be adapted. The size and the number of messages between sites is more important. A simple optimization is to compute selections on tables at the local sites.

10.4.3 Fragmentation of the database

Important design choices are made when the database is distributed over the available sites. To what extent should the information in the database be replicated? What is a good fragmentation of the information in the database? The answer to these choices depends on the topology and capacity of the sites that are cooperating to form the distributed database.

The problem to fragment a database in such a way that with a uniform access distribution the workload is optimally divided over the sites is NP-complete. It is therefore interesting to investigate what extra knowledge about the access behaviour of the database is needed to come up with good distributions of the database. An interesting starting points is for example knowledge about the access points of data items. If a data item is only accessed by users from one or two sites, it is natural to store the requested item on at least one of these sites.

The relation between fragmentation and replication can be studied. Of course, access times are optimal in the fully replicated case. However, if a lot of updates take place in the database, the replication of data introduces extra overhead to maintain correctness instead of speeding up transactions. To what extent a database should be replicated to provide the optimal access behaviour is an open question.

10.4.4 Synchronising sites

The scheduler can receive new transactions at more than one site. Existing (centralised) schedulers that make use of unique time-stamps have to ensure that time-stamps issued at different sites are unique and somehow related (thus time-stamps issued at roughly the same time should have roughly the same value).

In general, important execution steps of transactions should be synchronised over all sites. This means that algorithms have to make sure that all sites have finished such an important step before they proceed to the next step of the transaction. Examples are synchronisation between the locking and unlocking phase in two phase locking, and the commitment of transactions. Unpredictable results will follow if a transaction commits on one site and aborts at another site.

Adapted algorithms for distributed databases have been provided for most existing (centralised) database mechanisms. It is hard to provide algorithms that are efficient, reliable and offer graceful degradation of the database in case of failures, but there is already a large library of generic algorithms that provide good communication protocols between sites in a distributed network.

Bibliography

[GM83] Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM TODS vol. 8 pp 186,213*, 1983.

[GS85] N. Goodman and D. Shasha. Semantically-based concurrency control for search structures. *Proceedings of the ACM pag 8-19*, 1985.

[IK94] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relation joins. *ACM transactions on database systems pp 482-502*, 1994.

[JK84] M. Jarke and J. Koch. Query optimization in database systems. *ACM computing survey pp 111-152*, 1984.

[KM93] Tei-Wei Kuo and Aloysius Mok. Ssp: a semantics-based protocol for real-time data access. *Proceedings 14th real-time systems symposium*, 1993.

[Kna87] Edgar Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys 19(4) p. 303*, 1987.

[KR81] H.T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM 0362-5915/81/0600-0213*, 1981.

[KR92] Mohan Kamath and Krithi Ramamritham. Performance characteristics of epsilon serializability with hierarchical inconsistency bounds. Technical report, University of Massachusetts, 1992.

[ÖzsuV91] M. Özsu and P. Valduriez. *Principles of distributed database systems.* Prentice-Hall International Editions, 1991.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the association for computing machinery*, Oktober1979.

[SY82] M. Saccoo and S. Yao. Query optimization in distributed database systems. *Advances in computers, volume 21 pp 225-273*, 1982.

[Vid85] K. Vidiasankar. A simple characterization of database serializability. *5th conf. on foundations of software technology and theoretical computer sciency, LNCS 206*, 1985.

[Vid91] K. Vidiasankar. Unified theory of database serializability. *Fundamenta Informaticae XIV*, 1991.

[Wei87] William E. Weihl. Distributed version management for read-only actions. *IEEE transactions on software engineering No. 1*, 1987.

[YA88] Shyan-Ming Yuan and Ashok. K. Agrawala. A class of optimal decentralized commit protocols. *IEEE?*, 1988.

[YDL93] P. Yu, D. Dias, and S. Lavenberg. On the analytical modeling of database concurrency control. *Journal of the ACM p. 831-872*, 1993.

Index

Performance Analysis for Real-time Databases

Simone A.E. Sassen
Statistics and Operations Research Group
Dept. of Mathematics and Computing Science
Eindhoven University of Technology

February 23, 1995

Abstract

In this talk we focus on uncertainty in real-time databases. In most applications of real-time databases, no exact information is available about the arrival times, the sizes and the deadlines of future transactions. Due to this uncertainty it is not sufficient to measure the performance of a real-time database in terms of the *average* response time of a transaction; one has to obtain an approximation for its *distribution*. The ultimate result should be to find an approximation for the probability that a transaction meets its deadline.

Performance Requirements

As has been explained in the talk of Maarten P. Bodlaender, real-time databases have to satisfy both database and real-time requirements.
Database requirement:

- execution of transactions must preserve consistency.

Real-time requirements:

- some transactions can only be executed in a certain time interval
- transactions must meet their deadlines.

The purpose of the performance analysis for real-time databases (RTDB's) is to look at schedulers that preserve the consistency of the database and to investigate how well the real-time requirements are satisfied.

Performance questions that could be of interest are: (1) which percentage of the transactions meets its deadline, (2) what is the transaction throughput of the RTDB, (3) how reliable is the RTDB (how small is the probability of deadlock), and so on. Motives for an interest in question (1) are

- to offer a high customer service level
  (Rabobank, PTT Telecom, Ericsson)→ **90% ?**

- it is very important for the control of (physical) processes, where deadlines may be quite firm
  (ECT, research project at INRIA) → **99%** or even **99.99% ?**

To be able to give an answer to performance questions it is necessary to investigate the **response time** of a (type of) transaction in the system. Having a good approximation for the average response time and the standard deviation of the response time could already enable us to judge if the RTDB can meet a performance level of about 90%. However when one wants to distinguish a level of 99% from 99.9%, a very accurate approximation of the probability distribution of the response time is needed. It is not very likely that we will be able to find such an accurate approximation for the response time distribution, but the aim is to at least derive approximations that are good enough for RTDB design where performance levels of about 90% or 95% are required.

How to Model?

Uncertain factors that influence the response time distribution are:

1. **The arrival instants of transactions.**
   In a real-time environment it is usually not known beforehand when arrivals of transactions will take place. There may be some type of transaction that comes within regular (known) intervals; other transactions arrive irregularly, which has to be modelled as a stochastic process.

2. **Sizes of transactions** (# data items needed + amount of work).
   These may also vary per transaction (type). The more data items are needed, the more data contention can occur so the longer the response time of the transaction might be. The amount of work a transaction requires can vary since this involves e.g. communication times and computing times. A probability distribution for the size and the amount of work of the transactions is needed.

3. **Deadlines and priorities of transactions.**
   Jobs with high priority will increase the response time of lower priority jobs.

4. **Arrival locations of transactions** (in distributed databases).
   Which percentage of the transactions arrives at which location?

5. **Data-item popularity.**
   If there are some popular data-items that are used by a lot of transactions and (for example) 2-phase locking is used to protect the data, the response times of these transactions can grow quite large since these transactions must wait for the popular data-items to become available. If there are no popular data-items (uniform data-access) the influence of data contention on the response times might be very small.

6. **Transactions can trigger new transactions.**
   This would mean that the times between arrivals are not independent.

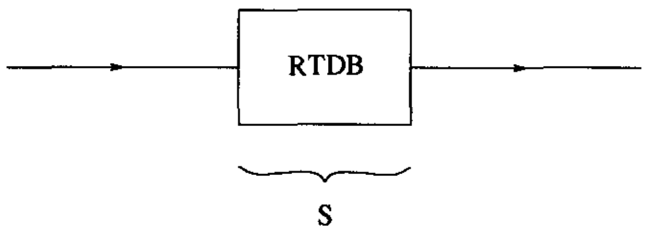


Figure 1: THE RTDB AS A BLACK BOX

How the above factors should be modelled depends on the specific application. To be able to formulate a good model for deriving an approximation for the distribution of the response time, it is vital to know which of the above elements play an important role in the various applications of the user group, and which elements can be ignored. For instance: is the data-access uniform, are the transactions of about the same size and type, what is the conflict probability between two transactions, and do transactions arrive regularly?
Hopefully this workshop and further discussions with the members of the user committee can contribute to getting a clear view of what the important elements are for each of the applications of the user committee.

Performance Analysis by Use of Queueing Models

A way of deriving approximations for the response time of transactions is by stochastic modelling. If we see the response time as a stochastic variable that is influenced by the elements mentioned in the previous paragraph, we can use queueing theory for approximating the response time distribution.

Queueing models have their origin in the study of design problems of automatic telephone exchanges and were first analysed by the queueing pioneer A.K. Erlang in the early 1900s. In the last 30 years, quite some progress has been made in the theory of queueing models. They have been applied e.g. to the design of computer systems, telecommunication networks and many problems in manufacturing. Our feeling is that they can be very useful for evaluating the performance of real-time databases.

The most abstract way of seeing a real-time database system is as in Figure 1. Transactions are entering the system, and after having spent a time S in the system they leave. What happens inside the 'black box' is not clear and will depend on the application. The response time S may consist of both **waiting time** (on locks or hardware) and **service time** (for the actual execution). For a good approximation of the response time a more detailed view of the database system (the black box) is needed. How many processors are available at which sites? Where are the data-items located? Which locking protocol is used? Modelling the database system as a queueing system requires information on when and where in the system transactions have to wait, and which service discipline is used.

Suppose we had only one type of transaction with fixed transaction size, and we made the following assumptions:

- The transactions arrive independently of each other.
- The time between 2 arrivals is exponentially distributed. In other words, the arrival process is a Poisson process.
  This assumption is often used in queueing models. Reasons are:
  1. it is an excellent approximation in the case of a large number of potential customers where each customer has a very small probability of arriving in a specific time interval (think of the 7 million people that have a telephone: each of them has a very small probability of making a phone call in a specific period of (say) 1 second);
  2. it has computationally attractive properties that simplify an analysis, e.g. the *memoryless* property.

  Note that the assumption of a Poisson arrival process may not be reasonable if there is a great regularity in the arrival of transactions.
- The service time of a transaction is exponentially distributed.
  This assumption is usually not a good approximation but because of the nice properties of the exponential distribution it is often taken to start with.

On the last page of this text, three representations of the real-time database system are given, based on the above assumptions.

I The top one is the most rigorous way of simplifying the database system and is easiest to analyse. The database can handle only one transaction at a time (the service discipline is FCFS), and all other transactions wait in a queue. For this model an exact expression exists for the distribution of the response time. When deadlines are taken into account by giving the transaction with the earliest deadline the highest priority, it is still possible to find (an approximation for) the distribution of the response time.

II The second figure releases the assumption of only one server. It assumes several parallel servers and thus can handle more transactions at a time. When more than k transactions are present, only k can be served and the remainder has to wait in a queue. For this queueing model an exact expression for the distribution of the response time is known. For generally distributed service times approximations for the response time distribution are available.
However when the locking protocol is taken into account, approximating the

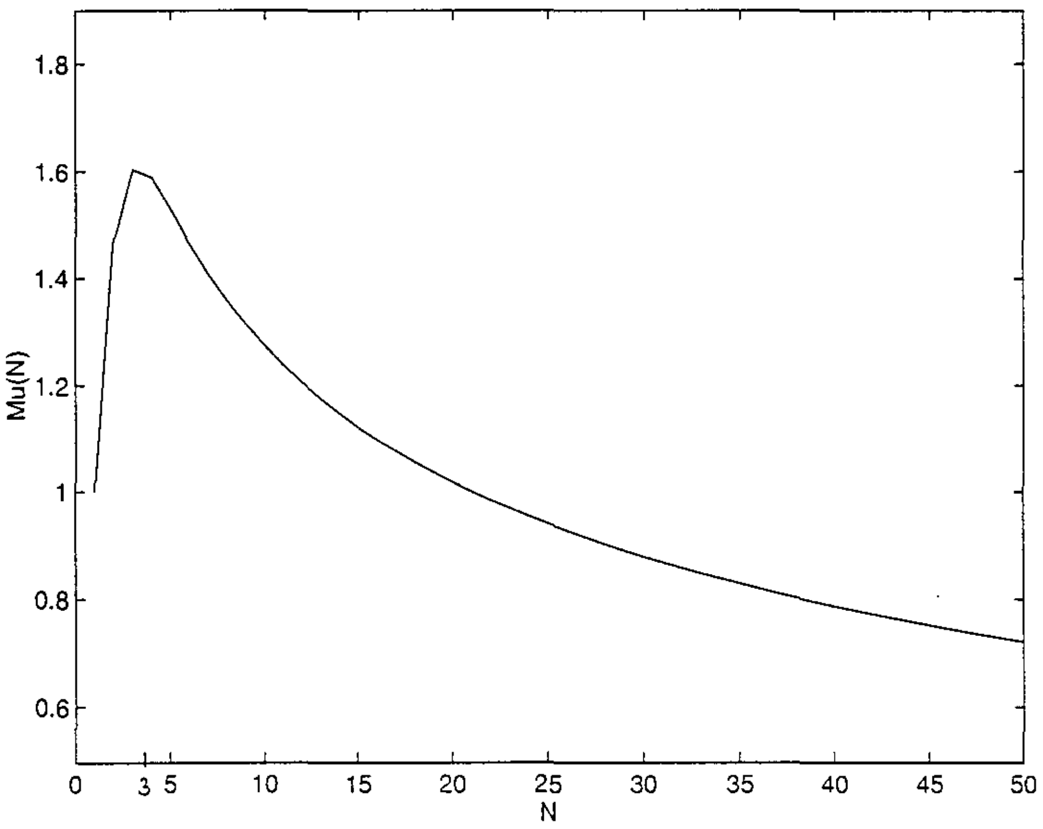
response time becomes more tedious. Because then a transaction that is executing at server 1 could have to wait for some data-item currently in use by the transaction executing at server k.

III The last figure shows a model for optimistic concurrency control that we have been investigating in more detail. It is assumed that each transaction demands some CPU-time (where the CPU is a single server that can handle only one transaction at a time, the rest has to wait in a queue), after which some computations have to be done at another site or processor (represented as an infinite server, so without capacity restrictions). After each visit to the CPU the transaction leaves the system with some probability or goes through another cycle with 1 minus that probability. A transaction that tries to leave enters a validation phase, in which it checks if a conflicting transaction committed during its execution. If no conflicting transaction committed during the execution of the validating transaction, the transaction can commit and leave the database system. Otherwise, the transaction goes back to the CPU and has to be rerun.

An approximate analysis of the system has resulted in the conclusion that the system is not ergodic, i.e. if the number of transactions allowed in the system is not restricted to some value N, there may be an infinite number of transactions cycling in the modelled system. Consequently, the performance of the system (the throughput of the system) decreases to zero as the number N of transactions allowed tends to infinity. For any choice of input values for the conflict probability and the speeds of the servers, it can be shown that this is the case.

The behaviour of the system is intuitively understandable, for the probability that a transaction has to be rerun depends on the number of transactions that has committed during its execution. Now, as the number of transactions in the system is larger, it is more likely that a conflicting transaction has committed during the execution of the transaction that tries to validate, so the higher is the probability that the validating transaction has to be rerun. This makes the system even fuller and eventually more transactions are entering the system per unit time than are leaving, resulting in a throughput that diminishes to zero.

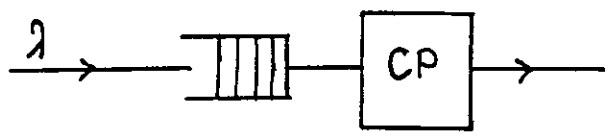
A typical illustration of the degrading performance is given in the plot below. Here we also see that there is a value of N for which the throughput of the system is maximal.



$\mu(N) \rightarrow 0$ for $N \rightarrow \infty$.

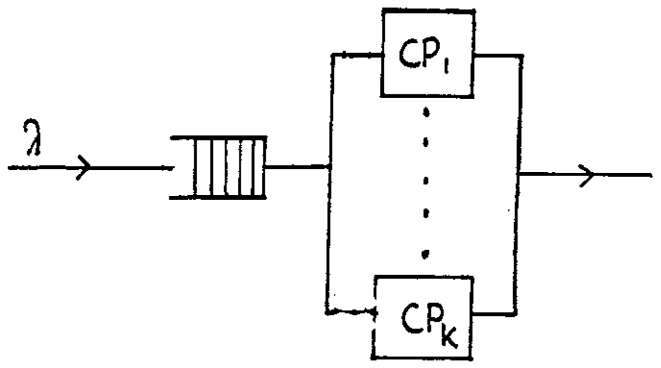
$\mu_1 = 2$
$p = 0.1$
$\lambda = 0.5$

I



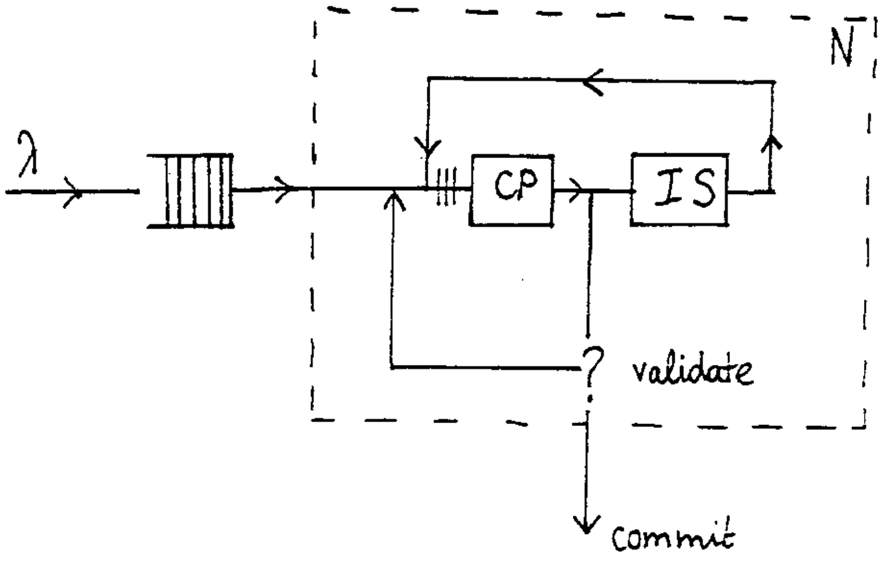
FCFS, 1 server

II



k parallel servers

III



- fixed transaction size
- p = Probability that 2 transactions conflict

93/31	W. Körver	Derivation of delay insensitive and speed independent CMOS circuits, using directed commands and production rule sets, p. 40.
93/32	H. ten Eikelder and H. van Geldrop	On the Correctness of some Algorithms to generate Finite Automata for Regular Expressions, p. 17.
93/33	L. Loyens and J. Moonen	ILIAS, a sequential language for parallel matrix computations, p. 20.
93/34	J.C.M. Baeten and J.A. Bergstra	Real Time Process Algebra with Infinitesimals, p.39.
93/35	W. Ferrer and P. Severi	Abstract Reduction and Topology, p. 28.
93/36	J.C.M. Baeten and J.A. Bergstra	Non Interleaving Process Algebra, p. 17.
93/37	J. Brunekreef J-P. Katoen R. Koymans S. Mauw	Design and Analysis of Dynamic Leader Election Protocols in Broadcast Networks, p. 73.
93/38	C. Verhoef	A general conservative extension theorem in process algebra, p. 17.
93/39	W.P.M. Nuijten E.H.L. Aarts D.A.A. van Erp Taalman Kip K.M. van Hee	Job Shop Scheduling by Constraint Satisfaction, p. 22.
93/40	P.D.V. van der Stok M.M.M.P.J. Claessen D. Alstein	A Hierarchical Membership Protocol for Synchronous Distributed Systems, p. 43.
93/41	A. Bijlsma	Temporal operators viewed as predicate transformers, p. 11.
93/42	P.M.P. Rambags	Automatic Verification of Regular Protocols in P/T Nets, p. 23.
93/43	B.W. Watson	A taxomomy of finite automata construction algorithms, p. 87.
93/44	B.W. Watson	A taxonomy of finite automata minimization algorithms, p. 23.
93/45	E.J. Luit J.M.M. Martin	A precise clock synchronization protocol,p.
93/46	T. Kloks D. Kratsch J. Spinrad	Treewidth and Patwidth of Cocomparability graphs of Bounded Dimension, p. 14.
93/47	W. v.d. Aalst P. De Bra G.J. Houben Y. Komatzky	Browsing Semantics in the "Tower" Model, p. 19.
93/48	R. Gerth	Verifying Sequentially Consistent Memory using Interface Refinement, p. 20.
94/01	P. America M. van der Kammen R.P. Nederpelt O.S. van Roosmalen H.C.M. de Swart	The object-oriented paradigm, p. 28.
94/02	F. Kamareddine R.P. Nederpelt	Canonical typing and Π-conversion, p. 51.
94/03	L.B. Hartman K.M. van Hee	Application of Marcov Decision Processe to Search Problems, p. 21.
94/04	J.C.M. Baeten J.A. Bergstra	Graph Isomorphism Models for Non Interleaving Process Algebra, p. 18.
94/05	P. Zhou J. Hooman	Formal Specification and Compositional Verification of an Atomic Broadcast Protocol, p. 22.
94/06	T. Basten T. Kunz J. Black M. Coffin D. Taylor	Time and the Order of Abstract Events in Distributed Computations, p. 29.
94/07	K.R. Apt R. Bol	Logic Programming and Negation: A Survey, p. 62.
94/08	O.S. van Roosmalen	A Hierarchical Diagrammatic Representation of Class Structure, p. 22.
94/09	J.C.M. Baeten J.A. Bergstra	Process Algebra with Partial Choice, p. 16.

94/10	T. verhoeff	The testing Paradigm Applied to Network Structure. p. 31.
94/11	J. Peleska C. Huizing C. Petersohn	A Comparison of Ward & Mellor's Transformation Schema with State- & Activitycharts, p. 30.
94/12	T. Kloks D. Kratsch H. Müller	Dominoes, p. 14.
94/13	R. Seljée	A New Method for Integrity Constraint checking in Deductive Databases, p. 34.
94/14	W. Peremans	Ups and Downs of Type Theory, p. 9.
94/15	R.J.M. Vaessens E.H.L. Aarts J.K. Lenstra	Job Shop Scheduling by Local Search, p. 21.
94/16	R.C. Backhouse H. Doornbos	Mathematical Induction Made Calculational, p. 36.
94/17	S. Mauw M.A. Reniers	An Algebraic Semantics of Basic Message Sequence Charts, p. 9.
94/18	F. Kamareddine R. Nederpelt	Refining Reduction in the Lambda Calculus, p. 15.
94/19	B.W. Watson	The performance of single-keyword and multiple-keyword pattern matching algorithms, p. 46.
94/20	R. Bloo F. Kamareddine R. Nederpelt	Beyond β-Reduction in Church's $\lambda\rightarrow$, p. 22.
94/21	B.W. Watson	An introduction to the Fire engine: A C++ toolkit for Finite automata and Regular Expressions.
94/22	B.W. Watson	The design and implementation of the FIRE engine: A C++ toolkit for Finite automata and regular Expressions.
94/23	S. Mauw and M.A. Reniers	An algebraic semantics of Message Sequence Charts, p. 43.
94/24	D. Dams O. Grumberg R. Gerth	Abstract Interpretation of Reactive Systems: Abstractions Preserving $\forall$CTL*, $\exists$CTL* and CTL*, p. 28.
94/25	T. Kloks	$K_{1,3}$-free and W_4-free graphs, p. 10.
94/26	R.R. Hoogerwoord	On the foundations of functional programming: a programmer's point of view, p. 54.
94/27	S. Mauw and H. Mulder	Regularity of BPA-Systems is Decidable, p. 14.
94/28	C.W.A.M. van Overveld M. Verhoeven	Stars or Stripes: a comparative study of finite and transfinite techniques for surface modelling, p. 20.
94/29	J. Hooman	Correctness of Real Time Systems by Construction, p. 22.
94/30	J.C.M. Baeten J.A. Bergstra Gh. Ştefanescu	Process Algebra with Feedback, p. 22.
94/31	B.W. Watson R.E. Watson	A Boyer-Moore type algorithm for regular expression pattern matching, p. 22.
94/32	J.J. Vereijken	Fischer's Protocol in Timed Process Algebra, p. 38.
94/33	T. Laan	A formalization of the Ramified Type Theory, p.40.
94/34	R. Bloo F. Kamareddine R. Nederpelt	The Barendregt Cube with Definitions and Generalised Reduction, p. 37.
94/35	J.C.M. Baeten S. Mauw	Delayed choice: an operator for joining Message Sequence Charts, p. 15.
94/36	F. Kamareddine R. Nederpelt	Canonical typing and Π-conversion in the Barendregt Cube, p. 19.
94/37	T. Basten R. Bol M. Voorhoeve	Simulating and Analyzing Railway Interlockings in ExSpect, p. 30.
94/38	A. Bijlsma C.S. Scholten	Point-free substitution, p. 10.

94/39	A. Blokhuis, T. Kloks	On the equivalence covering number of splitgraphs, p. 4.
94/40	D. Alstein	**Distributed Consensus and Hard Real-Time Systems, p. 34.**
94/41	T. Kloks, D. Kratsch	Computing a perfect edge without vertex elimination ordering of a chordal bipartite graph, p. 6.
94/42	J. Engelfriet, J.J. Vereijken	Concatenation of Graphs, p. 7.
94/43	R.C. Backhouse, M. Bijsterveld	Category Theory as Coherently Constructive Lattice Theory: An Illustration, p. 35.
94/44	E. Brinksma, J. Davies, R. Gerth, S. Graf, W. Janssen, B. Jonsson, S. Katz, G. Lowe, M. Poel, A. Pnueli, C. Rump, J. Zwiers	Verifying Sequentially Consistent Memory, p. 160
94/45	G.J. Houben	Tutorial voor de ExSpect-bibliotheek voor "Administratieve Logistiek", p. 43.
94/46	R. Bloo, F. Kamareddine, R. Nederpelt	The λ-cube with classes of terms modulo conversion, p. 16.
94/47	R. Bloo, F. Kamareddine, R. Nederpelt	On Π-conversion in Type Theory, p. 12.
94/48	Mathematics of Program Construction Group	Fixed-Point Calculus, p. 11.
94/49	J.C.M. Baeten, J.A. Bergstra	Process Algebra with Propositional Signals, p. 25.
94/50	H. Geuvers	A short and flexible proof of Strong Normalazation for the Calculus of Constructions, p. 27.
94/51	T. Kloks, D. Kratsch, H. Müller	Listing simplicial vertices and recognizing diamond-free graphs, p. 4.
94/52	W. Penczek, R. Kuiper	Traces and Logic, p. 81
94/53	R. Gerth, R. Kuiper, D. Peled, W. Penczek	A Partial Order Approach to Branching Time Logic Model Checking, p. 20.
95/01	J.J. Lukkien	The Construction of a small CommunicationLibrary, p.16.
95/02	M. Bezem, R. Bol, J.F. Groote	Formalizing Process Algebraic Verifications in the Calculus of Constructions, p.49.
95/03	J.C.M. Baeten, C. Verhoef	Concrete process algebra, p. 134.
95/04	J. Hidders	An Isotopic Invariant for Planar Drawings of Connected Planar Graphs, p. 9.
95/05	P. Severi	A Type Inference Algorithm for Pure Type Systems, p.20.
95/06	T.W.M. Vossen, M.G.A. Verhoeven, H.M.M. ten Eikelder, E.H.L. Aarts	A Quantitative Analysis of Iterated Local Search, p.23.
95/07	G.A.M. de Bruyn, O.S. van Roosmalen	Drawing Execution Graphs by Parsing, p. 10.
95/08	R. Bloo	Preservation of Strong Normalisation for Explicit Substitution, p. 12.
95/09	J.C.M. Baeten, J.A. Bergstra	Discrete Time Process Algebra, p. 20
95/10	R.C. Backhouse, R. Verhoeven, O. Weber	Math∫pad: A System for On-Line Prepararation of Mathematical Documents, p. 15

95/11	R. Seljée	Deductive Database Systems and integrity constraint checking, p. 36.
95/12	S. Mauw and M. Reniers	Empty Interworkings and Refinement Semantics of Interworkings Revised, p. 19.
95/13	B.W. Watson and G. Zwaan	A taxonomy of sublinear multiple keyword pattern matching algorithms, p. 26.
95/14	A. Ponse, C. Verhoef, S.F.M. Vlijmen (eds.)	De proceedings: ACP'95, p.
95/15	P. Niebert and W. Penczek	On the Connection of Partial Order Logics and Partial Order Reduction Methods, p. 12.
95/16	D. Dams, O. Grumberg, R. Gerth	Abstract Interpretation of Reactive Systems: Preservation of CTL*, p. 27.
95/17	S. Mauw and E.A. van der Meulen	Specification of tools for Message Sequence Charts, p. 36.
95/18	F. Kamareddine and T. Laan	A Reflection on Russell's Ramified Types and Kripke's Hierarchy of Truths, p. 14.
95/19	J.C.M. Baeten and J.A. Bergstra	Discrete Time Process Algebra with Abstraction, p. 15.
95/20	F. van Raamsdonk and P. Severi	On Normalisation, p. 33.
95/21	A. van Deursen	Axiomatizing Early and Late Input by Variable Elimination, p. 44.
95/22	B. Arnold, A. v. Deursen, M. Res	An Algebraic Specification of a Language for Describing Financial Products, p. 11.
95/23	W.M.P. van der Aalst	Petri net based scheduling, p. 20.
95/24	F.P.M. Dignum, W.P.M. Nuijten, L.M.A. Janssen	Solving a Time Tabling Problem by Constraint Satisfaction, p. 14.
95/25	L. Feijs	Synchronous Sequence Charts In Action, p. 36.
95/26	W.M.P. van der Aalst	A Class of Petri nets for modeling and analyzing business processes, p. 24.