Virtual Full Replication for Scalable Distributed Real-Time Databases

by

Gunnar Mathiason
Abstract

A fully replicated distributed real-time database provides high availability and predictable access times, independent of user location, since all the data is available at each node. However, full replication requires that all updates are replicated to every node, resulting in exponential growth of bandwidth and processing demands with the number of nodes and objects added. To eliminate this scalability problem, while retaining the advantages of full replication, this thesis explores Virtual Full Replication (ViFuR); a technique that gives database users a perception of using a fully replicated database while only replicating a subset of the data.

We use ViFuR in a distributed main memory real-time database where timely transaction execution is required. ViFuR enables scalability by replicating only data used at the local nodes. Also, ViFuR enables flexibility by adaptively replicating the currently used data, effectively providing logical availability of all data objects. Hence, ViFuR substantially reduces the problem of non-scalable resource usage of full replication, while allowing timely execution and access to arbitrary data objects.

In the thesis we pursue ViFuR by exploring the use of database segmentation. We give a scheme (ViFuR-S) for static segmentation of the database prior to execution, where access patterns are known a priori. We also give an adaptive scheme (ViFuR-A) that changes segmentation during execution to meet the evolving needs of database users. Further, we apply an extended approach of adaptive segmentation (ViFuR-ASN) in a wireless sensor network - a typical dynamic large-scale and resource-constrained environment. We use up to several hundreds of nodes and thousands of objects per node, and apply a typical periodic transaction workload with operation modes where the used data set changes dynamically. We show that when replacing full replication with ViFuR, resource usage scales linearly with the required number of concurrent replicas, rather than exponentially with the system size.

Keywords: Scalability, Flexibility, Adaptiveness, Database Replication, Resource Management, Distributed Database, Real-time Database.
There are many people involved in the process of writing a thesis. What may look like a one-person project is actually a mind-developing process, affecting many people. It has been very interesting to experience, and I’m very thankful to all the people who took part in it.

My advisor, Prof. Sten F. Andler: Thank you for your continuous and encouraging support, and for the many interesting discussions. Thank you for leveraging my skills in structuring and separating issues, and for teaching how to take new angles in problem analysis. My co-advisor, Prof. Sang H. Son: Thank you for all the positive feedback and encouragement. It has been great to share the path, ever since we first met in Providence, RI. Thank you also for inviting me to visit UVa during the spring of 2007. It was a great thesis booster. My co-advisor, Prof. Hans Hansson: Thank you for inspiration whenever we met, and for helping me wrap it all up. The CUGS graduate school and the ARTES network: I am also grateful for funding, for courses and events, and for giving a good network of contacts.

My colleagues of the DRTS group: Alexander, Ammi, Birgitta, Henrik, Jonas, Mats, Marcus, Robert, Ronnie and Sanny, - so many meetings, so much feedback! I often found that doing a presentation for you was more challenging and developing than the actual conference presentation. Also, many thanks to all my other great colleagues at the University of Skövde. My thanks also to other colleagues involved in the process: CUGS students and staff, UVa staff, Leo, and Woochul.

My close relatives: Parents, Margareta and Stig; parents-in-law, Ingalill and Sten; brothers, brothers-in-law and their families. In spite of never really understanding why and what this was about, you helped me in any way you could. It is good to have you around, folks.

Most importantly, my family: My wonderful wife, Lena, for her love and understanding, and for caring for us so well. Thank you for enduring too much time spent on work, and for sharing my working life as well. Our sons Jacob, Frederik, and Jesper: Thank you for understanding tired Dad, for taking care of Mum when I was away in Virginia, and for taking a lot of responsibility. That helped me focus. This thesis is dedicated to you, family.
List of Publications

Several contributions of this Ph.D. thesis have been previously published. The following publication list maps the primary publications to the contributions in the thesis. Contributions are fully described in Section 9.1. Any changes and elaborations of the work in these publications are listed as revisions.


  This paper presents the approach for using ViFuR in Wireless Sensor Networks (Contributions 10, 11 and 12). The material from this paper is mainly used in thesis chapter 7. No revisions.


  This paper fully elaborates the adaptive segmentation approach to ViFuR (Contributions 6, 7, 8 and 9), allowing changing access patterns to which segments adapt. The material from this paper is mainly used in thesis chapter 6. Revisions: Experiments for ViFuR-A were re-designed and re-executed, to make use of the replacement policy developed for ViFuR-ASN. Further, the load model was changed to better mimic a realistic application, and for a comparison with the load used for evaluation in chapter 7.

The Thesis Proposal integrates the work done up to this point. It elaborates on the background and the notation of segmentation. In addition, it defines the aims and subproblems to address for the thesis, as well as the methodology to use. The thesis proposal suggests a simulation study as a suitable approach for evaluation in a large-scale setting. The material from this thesis proposal is mainly used in chapters 2 and 3, while the methodology is applied through chapters 5, 6 and 7. Revisions: Subproblems are elaborated, and contributions in chapters 5, 6 and 7 are summarized and connected to subproblems in a Conclusions section of each chapter.


  This paper elaborates ViFuR using static segmentation, and describes an implementation approach for a table-based segmentation based on pre-specification of accesses (Contributions 1 partly, 2 and 3). Further, resource usage for such an approach is analyzed. The material from this paper is mainly used in chapters 4 and 5. Revisions: The sparse matrix representation for segmentation on multiple properties has been restructured for better scalability of representation. The usage of rules and the notation for rules have been elaborated.


  In this paper, the scalability problem of full replication is elaborated, and the segmentation approach is suggested for replica management (Contribution 1 partly). Further, the initial ideas for both static and adaptive segmentation are introduced. The material from this paper is mainly used in thesis chapters 3 and 4. No revisions
The following secondary publications are related to the primary publications in that they publish preliminary work that lead to other publications as a base for the thesis. These publications were developed as described below.


  This paper presents the testbed for the future work of evaluating the full implementation of ViFuR-ASN. It is a continuation of the paper "Virtual Full Replication for Scalable and Adaptive Real-Time Communication in Wireless Sensor Networks", published at the conference SENSORCOMM 2008.


  This paper introduces the two-tier approach of using a distributed real-time database as a whiteboard. The paper was later developed into the conference paper "Virtual Full Replication by Adaptive Segmentation", published at RTCSA 2007.


  This paper is the initial description of using simulation as an approach to study ViFuR in a large-scale distributed real-time databases, in a controlled environment. The work was integrated into the research method of the Thesis Proposal, also published in 2006.
# Contents

1 Introduction
   1.1 Application characteristics and examples . . . . . . . . . . . 3
   1.2 Approach . . . . . . . . . . . . . . . . . . . . . . . . . . 6
   1.3 Contributions . . . . . . . . . . . . . . . . . . . . . . . 7
   1.4 Limitations . . . . . . . . . . . . . . . . . . . . . . . . 8
   1.5 Thesis outline . . . . . . . . . . . . . . . . . . . . . . . 9

2 Background
   2.1 Distributed real-time databases . . . . . . . . . . . . . . . . . 11
      2.1.1 Database systems and transactions . . . . . . . . . . . 11
      2.1.2 Distributed database systems . . . . . . . . . . . . . . 13
      2.1.3 Real-time database systems . . . . . . . . . . . . . . 14
      2.1.4 Distributed real-time database systems . . . . . . . . 15
   2.2 The DeeDS database architecture . . . . . . . . . . . . . . . . 16
   2.3 Database model . . . . . . . . . . . . . . . . . . . . . . . 18
   2.4 Virtual Full Replication . . . . . . . . . . . . . . . . . . . 18
   2.5 Scalability and resource usage . . . . . . . . . . . . . . . . . 19
   2.6 Durability and Incremental diskless recovery . . . . . . . . . 23
   2.7 Simulation-based studies of systems . . . . . . . . . . . . . . 24

3 Problem Definition
   3.1 Problem introduction . . . . . . . . . . . . . . . . . . . . . . 31
      3.1.1 Scalability in fully replicated databases . . . . . . . . 32
      3.1.2 Allocation of data object replicas . . . . . . . . . . . 33
   3.2 Problem statement . . . . . . . . . . . . . . . . . . . . . . 34
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3</td>
<td>Problem decomposition</td>
<td></td>
<td>36</td>
</tr>
<tr>
<td>3.4</td>
<td>Assumptions</td>
<td></td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Research methodology</td>
<td></td>
<td>39</td>
</tr>
<tr>
<td>4</td>
<td>A Segmentation Approach for Virtual Full Replication</td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>A perception of full replication</td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>4.2</td>
<td>Database segmentation</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Object properties and segments</td>
<td></td>
<td>43</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Segmentation</td>
<td></td>
<td>44</td>
</tr>
<tr>
<td>4.2.3</td>
<td>A segmentation example</td>
<td></td>
<td>46</td>
</tr>
<tr>
<td>4.3</td>
<td>Segmentation on multiple properties</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Using multiple properties of database objects</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Related object properties</td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Forming and using rules for related object properties</td>
<td></td>
<td>52</td>
</tr>
<tr>
<td>4.4</td>
<td>An architecture for segmentation</td>
<td></td>
<td>56</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Model design of a database node</td>
<td></td>
<td>58</td>
</tr>
<tr>
<td>4.5</td>
<td>Conclusions</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>4.5.1</td>
<td>Results, contributions and subproblems met</td>
<td></td>
<td>66</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Discussion</td>
<td></td>
<td>67</td>
</tr>
<tr>
<td>5</td>
<td>Static Segmentation for Fixed Object Allocations</td>
<td></td>
<td>69</td>
</tr>
<tr>
<td>5.1</td>
<td>Implementing static segmentation</td>
<td></td>
<td>70</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Static segmentation on multiple properties</td>
<td></td>
<td>73</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Applying rules for property consistency</td>
<td></td>
<td>74</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Multiple segmentations</td>
<td></td>
<td>75</td>
</tr>
<tr>
<td>5.2</td>
<td>Efficient coding of segments using Bloom Filters</td>
<td></td>
<td>76</td>
</tr>
<tr>
<td>5.3</td>
<td>Distribution of static segments</td>
<td></td>
<td>78</td>
</tr>
<tr>
<td>5.4</td>
<td>Analysis</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>5.4.1</td>
<td>The cost and benefits of segmentation</td>
<td></td>
<td>80</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Resource usage in a segmented database</td>
<td></td>
<td>81</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusions</td>
<td></td>
<td>86</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Results, contributions and subproblems met</td>
<td></td>
<td>86</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Discussion</td>
<td></td>
<td>86</td>
</tr>
</tbody>
</table>
6 Adaptive Segmentation for Dynamic Object Allocations 89
   6.1 The need for adaptive segmentation .................. 90
   6.2 ViFuR-A ........................................... 92
       6.2.1 Adaptive segment management ................. 93
       6.2.2 Object directory .......................... 96
       6.2.3 Specification of data requirements .......... 98
       6.2.4 Replica establishment ......................... 99
       6.2.5 Replica removal ................................ 102
       6.2.6 Removal preventsions .......................... 104
       6.2.7 Removal decision policy ....................... 105
       6.2.8 Concurrency issues in segment adaptation .... 108
   6.3 Evaluation of ViFuR-A ................................ 110
       6.3.1 S1: Comparing to alternative replication strategies . 114
       6.3.2 S2: Scalability and adaptation ............... 120
   6.4 Conclusions ........................................ 128
       6.4.1 Results, contributions and subproblems met .................. 129
       6.4.2 Discussion .................................. 130

7 Using Virtual Full Replication 131
   7.1 Using ViFuR in wireless sensor networks .............. 132
       7.1.1 Issues with wireless sensor networks ............ 132
       7.1.2 Comparing ViFuR for the LAN and WSN environ-
           ments ........................................... 134
   7.2 ViFuR-ASN ........................................... 140
       7.2.1 A database approach for communication in WSN .... 140
       7.2.2 Database tier .................................. 142
       7.2.3 Sensor tier .................................... 143
       7.2.4 Establishment and removal of replicas .......... 144
   7.3 Evaluation of ViFuR-ASN ................................ 149
       7.3.1 Simulation model ................................ 149
       7.3.2 Simulation settings ............................ 150
       7.3.3 S1: Comparing to alternative strategies .......... 151
       7.3.4 S2: Scalability and adaptation .................. 155
   7.4 Conclusions ........................................ 159
Chapter 1

Introduction

_I may not have gone where I intended to go,
but I think I have ended up where I intended to be._
- Douglas Adams

In a distributed database, performance and availability can be improved by allocating data at the nodes where data is mostly used. With real-time databases, transaction timeliness is a major concern and there is a need for performance guarantees. By allocating a replica of data to every node where the data might be used, transactions do not need to access data remotely over the network, and transaction timeliness becomes independent of delays in the network.

In a _fully replicated_ database, the entire database is available at all the nodes. With all data locally available, the user need no remote processing to use data. Full replication supports performance at the cost that some local data replicas may never be used. Considering typical usage of data, full replication uses an excessive amount of system resources, since the system must replicate all the updates, written to any replica, to all the nodes. This causes a scalability problem in the usage of bandwidth for the replication of updates, the utilization of storage for data replicas, and the use of processing for replicating updates and resolving conflicts for concurrent and conflicting updates.
In this thesis we explore and evaluate Virtual Full Replication (ViFuR) (Andler, Hansson, Eriksson, Mellin, Berndtsson & Eftring 1996, Mathiason & Andler 2003) as an approach to give the database user a perception of full replication, while replicating only what is needed for such a perception. This enables availability and timeliness without the scalability problem of full replication. The database is segmented into groups of data objects that share some properties. This enables the individual allocation of objects and segments, as well as a scalable resource management approach. We present how segments are formed and used, and provide algorithms and a model architecture for the coexistence of multiple replication methods. In addition, we present and evaluate adaptive and incremental change to segments for allocation, such that scalability is maintained over time by allocating data object replicas only to the nodes where needed. With our approach, resource usage scales with the actual need for replicas rather than the scale of the system. This means that for a typical application, resource usage will increase more slowly when the system is scaled up. Finally, we apply and evaluate the approach in a typical large-scale resource-constrained distributed system, a Wireless Sensor Network.

The ViFuR approach is very suitable for a distributed real-time main memory database with eventual consistency, where local data availability is essential for timeliness of the real-time application. The DeeDS database prototype (Andler et al. 1996) is such a database system which stores a fully replicated database entirely in main memory for independence of disk accesses, in order to enable transaction timeliness. Using full replication together with detached replication allows transactions be to executed on the local node, independent of any network delays. All replicas are primary replicas that can receive updates, also concurrently. Detached replication propagates updates to other nodes after transaction commit, independently from the execution of the transaction. With full replication of data objects and detached replication of updates, database clients get the perception of a single local database, such that they are location-unaware and need not to synchronize concurrent updates to different replicas of data objects. In DeeDS, all data replicas are primary replicas that can be updated concurrently at any node. The PRiDe replication protocol (Syberfeldt 2007) ensures
1.1 Application characteristics and examples

that possible update conflicts are found and resolved.

Many approaches for distributed real-time databases often use distrib-
uted transactions and do not utilize relaxed consistency (Ozsu & Valduriez
1991), or they use a single primary copy where all updates must be pro-
cessed. Such approaches suffer from resource-demanding replica synchro-
nization, which do not allow large-scale systems to be built. However, the
need for real-time systems with large-scale data distribution increases. In
this thesis we show how resource usage in a distributed real-time database
can be bounded by considering the typical data usage, in order to achieve
scalability while still having the flexibility of a fully replicated database,
such as DeeDS.

1.1 Application characteristics and examples

The ViFuR schemes presented in the thesis aim to be used with large-scale
distributed systems where local real-time characteristics is important for the
database application, and applications that can use a data-centric communi-
cation approach. We apply ViFuR in the context of large scale distributed
real-time databases. In section 9.2 we list application profiles that collect
characteristics of typical applications that benefit from the ViFuR approach.
To summarize, ViFuR is suitable for applications with the following char-
acteristics:

- Large-scale resource-constrained data-centric distributed systems
  with requirements of timely access to data. Data is published for all
  the nodes, while data is used by a few nodes concurrently. A major-
  ity of distributed applications need only a few replicas concurrently.
  ViFuR is not suited for systems that require full replication, since it
  uses additional resources, in both bandwidth and storage, for the man-
  agement of replication.

- Distributed systems where nodes are added and removed during exe-
  cution, and where a certain configuration of nodes is in use during
  a period of operation and then reconfigures. Using ViFuR in sys-
  tems where reconfiguration periods have short periods, approaching
the period of data accesses, requires a high amount of adaptation processing. Typical real-time applications have modes of operations for periods of time which are typically much longer than the periods of data access.

- Applications with a high degree of cohesive data, that is, groups of data objects that are closely used, typically accessed as an entire group of data, or with access periods for the group that coincide in time. Smaller cohesive data groups and groups that are shared between a small set of nodes benefit more from ViFuR. Applications with wide groups, or with no cohesiveness will generate many replicas with our scheme, and the benefit of using it diminishes.

The key benefit of ViFuR is that large-scale distributed real-time applications with timely transactions can be built. Alternative approaches often use explicit static configuration of communication and replication, or central and resource-demanding solutions. Three examples of typical applications that are enabled for scalability with ViFuR are: A car system with a distributed set of ECUs connected by an in-vehicle network such as CAN, a Wireless Sensor Network (WSN), and a Communication backbone for a wildfire fighting mission.

1. The car system of distributed processors (ECUs) is a very common example of a system with several control processors using a homogeneous network. Loosely coupled processors have individual, time-critical tasks to perform for the operation of the overall system, and where processors share data to perform their individual tasks and for the operation of the car. An example is the gearbox and brake system ECUs, which can improve their local control by using data from the ignition ECU, about the number of revolutions of the engine. Such an exchange between ECUs is typically done by an ad-hoc and explicit declaration of data exchange (Nyström, Tesanovic, Norström, Hansson & Bånkestad 2002). A distributed real-time database with ViFuR used as a whiteboard allows scalability, local timeliness, and simplifies ECU programming and configuration of data exchange.
1.1 Application characteristics and examples

2. A WSN is a popular application that uses a large set of nodes with limited communication, processing capability and storage. A typical Crossbow MICAz mote is limited to the bandwidth of 250 kB/S. It has 128 kB of program memory, and it is an 8-bit processor running at 8 MHz. The main memory used is a few tens of kB, while the measurement logging storage is up to 1 MB. Such limited nodes are typically battery-operated, with radio transmissions as the most energy-demanding task. Sensors monitor the environment and often transmit events timely as they occur. Also, clients need timely access to sensor data from different locations in the network. Further, communication often needs to connect nodes over many hops, involving many processors for the transfer of a single update, often all the way to the edge of the network. Such communication is a high-latency, low-bandwidth, and energy-consuming task. In practice, only a few hops can be used before communication breaks down. We show that by using a multi-tier distributed real-time database, motes can be offloaded from the resource-demanding task of multi-hop communication, while a ViFuR allows scalability and enables access to any sensor data at any node of the network (Chapter 7).

3. A wildfire fighting scenario is an example of an emergent embedded application in emergency management. We assume that rescue teams in the wildfire fighting scenario are equipped with wearable computers that can communicate. In addition, autonomous vehicles can be sent into especially dangerous areas. Each actor shares its view of the local surroundings with other actors through a distributed real-time database, and each actor has access to local as well as (a suitable subset of) global information. Particular subsets of the actors may also share specific information of mutual interest. Each actor needs timely access to task-critical data, such as updates of the status of open retreat paths, remaining supplies, and so on. A distributed database has been proposed by (Tatomir & Rothkrantz 2005) as a suitable infrastructure for emergency management. We believe that using a distributed real-time database with ViFuR as whiteboard communication between nodes can be used for scalable publishing and structured storage, for implicit
consistency management between replicas, achieving fault-tolerance, as well as for higher independence of user location by lowering the coupling between data clients. There is no need to explicitly coordinate communication in such distributed applications, which reduces complexity, in particular where actors communicate dynamically.

1.2 Approach

In this thesis we explore and develop the concepts needed for ViFuR, in order to enable scalability, flexibility, and transaction timeliness in large-scale distributed real-time databases. Inspired by principles used in caching, database buffering, and also virtual memory, we elaborate approaches to provide both a perception of full replication and its key advantages for timeliness.

- We assess how to manage individual degrees of replication for database objects, and present details of how segmentation provides a scalable approach for such object management. Further, we use the approach for static segmentation on pre-specified data object properties (ViFuR-S), where the property of allocation to nodes is the most important property. A static segmentation for allocation gives an optimal replication schema, in terms of local availability, for the set of accesses of a database application. A full pre-specification of accesses can be directly translated to an allocation schema that matches the local data needs. In the thesis, we analyze scalability by the resource usage of three key metrics and key resources: bandwidth, storage, and processing.

- We elaborate static segmentation by adaptive segmentation where accesses that cannot be pre-specified are taken into account to incrementally update segments during execution time, such that a near-optimal replication schema is maintained over time, also when mode changes of the database application result in changed data needs at the nodes. This approach is evaluated for scalability by resource usage of three key metrics: bandwidth, storage, and transaction processing delays.
1.3 Contributions

- We refine the adaptive approach for usage with a WSN, in which we evaluate and validate scalability in terms of bandwidth, storage, and transaction processing delay. A WSN is a typical current large-scale resource-constrained application, where a distributed real-time database simplifies communication and enables scalability.

1.3 Contributions

- We elaborate Virtual Full Replication by segmentation as an approach for scalable and flexible whiteboard communication, using a distributed real-time database. In order to do this, we formally define both ViFuR and segmentation, and show how segments are formed by using object properties, also for multiple segmentations on the same object set, as well as segmentations on combinations of object properties. A model architecture is also provided where multiple segmentations and multiple consistency classes can coexist.

- An efficient and scalable algorithm is presented for static segmentation of a database, based on pre-specification of accesses through transactions executed by the application, for hard and soft real-time database applications. This algorithm has $O(o \log o)$ computational complexity and $O(o + s)$ storage complexity for $o$ objects and $s$ segments for each segmentation, and where multiple segmentations can be generated for different purposes on subsets of properties.

- We give a distributed protocol with a name service (directory). This protocol manages incremental changes to segments such that new replicas can be established and unused replicas can be removed concurrently, based on current needs for data at each individual node. We present a generic deallocation mechanism that uses two parameters only, for the configuration of a generic access pattern that is sporadic. The scheme is evaluated using a detailed large-scale database system simulation.

- A novel two-tiered approach is provided for whiteboard communication in WSNs that enables scalability and gives location-independence
for data users in a WSN. The resource-constrained large-scale environment of a WSN using heterogeneous communication links is a well-motivated test bed for ViFuR, and it benefits to a great extent from the scalable and adaptive allocation of distributed real-time data. The scheme is evaluated by simulation.

• In our exploration of using ViFuR for scalability and flexibility, we see that applications with certain properties benefit more than other applications. These properties are condensed into application profiles, which serve as guidelines for choosing applications that benefit from ViFuR.

1.4 Limitations

ViFuR is developed for a distributed real-time database with replicas that are eventually consistent, and applications using such a database need to be tolerant to eventual consistency. However, the adaptive allocation of replicas is an approach that can be applied in distributed systems as a caching approach in general, for performance improvements due to improvement of local availability.

A central underlying assumption is that an actual workload or application will require only a few replicas of an objects concurrently at different nodes. For applications that need many replicas of a large share of the database, the resource usage will increase exponentially with the required number of replicas. For a system that requires full replication, ViFuR uses more resources than a fully replicated system without ViFuR.

The adaptiveness of replica allocations relies on that data accesses follow access patterns. In this thesis, we use periodic access patterns with mode changes as a generic access pattern that is common for real-time systems. ViFuR may also be used with more elaborate access patterns for applications where such access patterns are known. Since our focus is to develop a generic approach, we have not developed deallocation policies based on elaborate access patterns for specific applications, or for specific access pattern types.
1.5 Thesis outline

Chapter 2 presents the background and introduces to the area, while chapter 3 lays out the problem and its parts, Chapter 4 elaborates on Virtual Full Replication as a concept and introduces segments and how they are formed. Chapter 5 presents static segmentation, used in the ViFuR-S scheme, and Chapter 6 presents adaptive segmentation, used in the ViFuR-A scheme. In chapter 7, the use of adaptive segmentation for communication in wireless sensor networks using the ViFuR-ASN scheme is examined.
Chapter 2

Background

Nanos gigantum humeris insidentes.¹
- Bernard of Chartres

This chapter presents a background to the challenges of distributed real-time database systems and scalability. In such a database, data objects are allocated to different nodes, while real-time properties need to be satisfied. The chapter also introduces simulation-based evaluations of computer systems.

2.1 Distributed real-time databases

2.1.1 Database systems and transactions

A *database* is a related collection of data and meta-data. Meta-data is the information about the data collection, such as descriptions of the relations and data representation types used, or properties about the data itself. Databases are accessed by using queries for retrieving data and updates for storing data. Queries and updates use *transactions* for grouping the database operations that logically belong together, such that these operations are executed atomically, and that the transaction ensures a well-defined state of

¹“Dwarfs standing on the shoulders of giants.”
the database after the execution. Using transactions ensures that integrity constraints between data entities are preserved, such that data entities are consistent with the state of what is being represented in the environment. In addition to consistency with the environment, there are several other types of consistency, such as that between replicas of the same logical data object, and between different data objects in the database. Further, temporal validity influences the consistency of data. Data objects must have values that agree about the environment at the point in time when they are used together. Temporal validity is typically expressed as a time period when the value of a data object is valid for use. Consistency of data is often specified to be the correctness criterion for many database applications. If replicas of data in the database are not fully consistent at all times, the database cannot be used. However, many applications can tolerate temporary inconsistencies without being incorrect, since such systems can find and compensate for states resulting from using temporarily inconsistent values.

The term transaction is often given one of the following meanings (Gray & Reuter 1993): 1) The request or input message that started the operation (request/reply). 2) All effects of the execution of the operation (transaction). 3) The program(s) that execute(s) the operation (transaction program). ACID (Atomicity, Consistency, Isolation and Durability) properties of transactions guarantee the effect of the transaction and that its operations are dependable. ACID properties are (Gray & Reuter 1993): Atomicity - The changes made to the database by the transaction operations are atomic, i.e., either all changes or no changes apply; Consistency - A transaction does not violate any integrity constraints when transforming the state of the database, from one consistent state to another consistent state; Isolation - Transactions may execute concurrently, but a transaction never perceives that other transactions execute concurrently. That is, ongoing transactions are not observable, a transaction appears to execute on its own, in isolation; Durability - Once the transaction has been successfully completed, the changes are permanent and will not be deleted by subsequent failures.

Using transactions in a concurrent system simplifies the design of such a system, since transactions execute entirely in isolation, and it is not necessary to explicitly synchronize processes that access the same data. For
2.1 Distributed real-time databases

Distributed systems, transactions offer a way to abstract concurrency control and reduce the need for synchronization mechanisms acting between separated parts of the system.

2.1.2 Distributed database systems

Burns & Wellings (1997) define a distributed system as a system of multiple autonomous processing elements (nodes), cooperating for a common purpose. It can either be a tightly or loosely coupled system, depending on whether the processing elements have access to a common memory or not.

A distributed database is a database allocated to multiple nodes in a distributed system, where the database is the object of distribution. The parts (the partitions) of the distributed database together form a logical database. With any distributed system, the partitioning of the application must be carefully considered. When distributing a database, we may allocate a partition (close) to a node where the data is most frequently used, which improves the availability of the data and increases the performance of the database, since network communication can be reduced. With such distribution, bandwidth requirement decreases while the overall system performance increases. Distributed transactions (Hevner & Yao 1979) are used to access data in a distributed database where data does not reside at the node of transaction instantiation. Such transactions are transferred to the nodes with the data needed, such that the actual execution of the transaction, as well as the data that the transaction uses, is distributed.

The cost of data accesses can be much lower with a distributed database than with a centralized database where all the data is accessed at a single node that is a single point of failure for the system. In addition, distribution is also an approach with which to overcome resource limitations, since it may divide the workload over multiple nodes. High availability of data is a key to performance in a distributed database. The distribution of data in a distributed real-time database is often a trade-off problem. With many replicas of the data, availability is high and read-accesses have low communication and delay costs. Unfortunately, in a distributed database with many replicas, update-accesses are expensive, since all updates must be sent to all the replicas. Further, with multiple replicas of the same data allocated to
different nodes, fault tolerance is improved, since a node may crash while its
data remains as a replica at some other node. There are several challenges in
a distributed database, including the dependency on communication links,
the consistency problems caused by delays of updates sent between nodes,
and the cost of agreement coordination of updates. Many approaches exist
in the literature for the optimal allocation of data, considering some cost
model for the distribution of the data over a network. The allocation and
management of replicas in distributed systems is a classic problem that is an
NP-complete (The File Allocation Problem)(Casey 1972), and that is typi-
cally approached by some near-optimal heuristics (Chandy & Hewes 1976).

2.1.3 Real-time database systems

Correct computer systems are expected to give a correct logical result from a
computation. In addition to such correctness, real-time systems are expected
to produce results in a timely fashion. Timeliness requirements are typically
expressed as deadlines, which specify when computing results are expected
to be available for usage. Several classifications exist for real-time systems.
One established classification is based on the value of the deadline. The
value includes both the benefits and the penalties of the timeliness of the
transaction. Deadlines may be hard, firm or soft, depending on the value of
the computation result if a deadline is missed (Locke 1986). Missing a hard
deadline has a large or infinite penalty, while a firm deadline miss gives no
value. For a soft deadline miss, there might still be some value from the
computation for some time.

A real-time database system needs timely data access, so that specified
access deadlines are met, and transactions in a real-time system need to be
time-cognizant (Ramamritham & Chrysanthis 1996). A transactions that
executes outside of the deadline boundaries has less value or may dam-
age the system, depending on the type of deadline associated with it. For
real-time databases, as with any real-time system, the most important real-
time characteristic is predictability. For real-time databases, predictability is
often more important than consistency, such that the consistency constraint
is relaxed for improved predictability of data accesses.
2.1 Distributed real-time databases

2.1.4 Distributed real-time database systems

In distributed real-time databases with distributed transactions, the transaction timeliness depends on the communication links and the delays of processing the transactions at the remote node. Transaction timeliness can be guaranteed only if the resources involved in the processing of the entire distributed transaction are known. In order to ensure timeliness, detailed a priori knowledge about the transactions’ resource requirements are necessary, including the worst case execution order of concurrent transactions, where the highest resource usage occurs. Resource requirements from certain critical execution orders, or critical transactions, must be known, so that the maximum resource needs can be specified. However, often far from all the requirements are fully known. A full analysis of the application is often difficult to make, thus unspecified overloads may still cause unpredictable delays to transactions. To ensure timeliness, it is therefore necessary to pessimistically pre-allocate resources for a worst case assumption on load, which typically lowers the efficiency dramatically for the system.

One approach to reduce the uncertainties of the specification is to remove sources of unpredictability involved in transaction execution, such as network delays and dependence on other nodes. Such sources are: 1) Disk access. Most databases have their persistent storage on hard disks, for which access times may be hard to bound. It is possible to define an average access time, but for real-time systems, the worst-case access time is what influences real-time behavior. For this reason, real-time databases may be implemented as main memory databases to enable predictable access times (García-Molina & Salem 1992). 2) Remote data access. Most commercial computer networks are built to support safe file transfers at best effort, where real-time properties are of less interest. Timeliness of remote accesses can be arbitrarily delayed. Some network types are very efficient (e.g. LANs), but worst-case communication delay times are very hard to bound. By using real-time network protocols, propagation time for messages can be bounded (Le Lann & Rivierre 1993). Another approach to avoid network delays is to allocate data at the node of execution. With full replication of the complete database at each local node, there is no need for unpredictable remote accesses. 3) Concurrent updates. In addition to the unpredictabil-
ity of the network, a remote access may be delayed by a concurrent data access or concurrent processing at the remote node. Interfering data accesses may be initiated at the remote node or even come from another remote node. By trading off consistency and allowing controlled temporary inconsistencies between replicas, independent updates to different replicas of the same data objects can be enabled. With local availability of data combined with detachment of any remote operation from the local execution of the transaction, the transaction can commit all operations locally before any network communication takes place, making the local execution time fully independent from any remote execution or update. Since the transaction is committed locally, only local worst-case transaction processing time needs to be analyzed and bounded. However, local-only commit protocols require conflict resolution mechanisms, such as version vectors (Parker & Ramos 1982) or generations (Syberfeldt 2007), to find and also resolve conflicts between independent updates that concurrently have updated replicas of the same data object at different nodes. 4) Failing nodes. Nodes with replicas in use may fail and replicas be destroyed. Failing nodes must be detected and recovered within a bounded time, shorter than the deadline of the timely transaction depending on the replica (Leifsson 1999).

2.2 The DeeDS database architecture

For predictability, the distributed real-time database system DeeDS (Andler et al. 1996) stores its database entirely in main memory, avoiding disk I/O delays caused by unpredictable access times for hard drives. There is no disk storage for durability. Instead, nodes act as peer backups for each other. To avoid transaction delays due to unpredictable network delays, the database is virtually fully replicated to all nodes, such that local database object replicas for transactions that execute are always available. This makes transaction execution timely and independent of network delays and network partitioning, since there is no need for remote data access during the execution of transactions. A (virtually) fully replicated database with detached replication, where replication is done after local transaction commit, allows independent updates, that is, concurrent and unsynchronized updates to replicas.
2.2 The DeeDS database architecture

of the same data object (Ceri, Houtsma, Keller & Samarati 1995). Such independent updates may cause database replicas to become inconsistent, and inconsistencies need to be resolved in the replication process by a conflict detection and resolution mechanism.

In DeeDS, update replication is detached from transaction execution, by propagation after transaction commit, and integration of replicated updates performed at all the other nodes. Conflicting updates are resolved at integration time. Temporary inconsistencies are thus allowed and also guaranteed to be eventually resolved, giving the database the property of eventual consistency (Definition 2.1) (Birrell, Levin, Needham & Schroeder 1982) (Saito & Shapiro 2005). Applications that use eventually consistent databases need to be tolerant of the temporarily inconsistent replicas, which can be achieved for many distributed and embedded applications.

**Definition 2.1.** Two different replicas of the same database object are eventually consistent, if they stabilize into a globally consistent state within a bounded number of processing steps, in a system that becomes quiescent.

In a (virtually) fully replicated database using detached replication, a number of predictability problems that are associated with the synchronization of concurrent updates at different nodes can be avoided, such as agreement protocols or the usage distributed locking of the replicas of objects, as well as reliance on stable communication to access data. Furthermore, the application programmer may assume that the entire database is available and that the application program has exclusive access to it. In addition, if the network of database nodes becomes partitioned, the users of the database can continue to execute transactions, since replicas of all used data are available locally. Conflicts that may be introduced during such partitioning are ensured to be resolved at re-connection, by the conflict detection and conflict resolution protocol PRiDe (Syberfeldt 2007). With this consistency management protocol, all replicas of the database are primary replicas that can be updated, and there is no single master replica of an object.
2.3 Database model

In the thesis, we use the following database model (Syberfeldt 2007). A database maintains a finite set of logical data objects $O = \{o_0, o_1, \ldots\}$, representing database values. Object replicas are physical manifestations of logical objects. A distributed database is stored at a finite set of nodes $\mathcal{N} = \{N_0, N_1, \ldots\}$. A replicated database contains a set of object replicas $\mathcal{R} = \{r_0, r_1, \ldots\}$. The function $R: O \times \mathcal{N} \rightarrow \mathcal{R}$ identifies the replica $r \in \mathcal{R}$ of a logical object $o \in O$ on a node $N \in \mathcal{N}$ if such a replica exists. $R(o, N) = r$ if $r$ is the replica of $o$ on node $N$. If no such replica exists, $R(o, N) = \text{null}$. A distributed database (or simply database) $D$ is a tuple $<O, \mathcal{R}, \mathcal{N}>$, where $O$ is the set of objects in $D$, and $\mathcal{R}$ is the set of replicas of objects in $O$, and $\mathcal{N}$ is the set of nodes such that each node $N \in \mathcal{N}$ hosts at least one replica in $\mathcal{R}$, i.e. $\mathcal{N} = \{N | \exists r \in \mathcal{R}(\text{node}(r) = N)\}$.

We model transaction programs, $T$, with two sets: The set of objects read by the transaction program, $\mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D}_T$ (the read set); The set of objects written by the transaction program, $\mathcal{W} \mathcal{R} \mathcal{I} \mathcal{T} \mathcal{E}_T$ (the write set). With this notation, a transaction program $T$ can be defined as $T = \mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D}_T, \mathcal{W} \mathcal{R} \mathcal{I} \mathcal{T} \mathcal{E}_T$. Also, we refer to the size of the read set as $r_T = |\mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D}_T|$, the size of the write set as $w_T = |\mathcal{W} \mathcal{R} \mathcal{I} \mathcal{T} \mathcal{E}_T|$. The working set $\mathcal{W} \mathcal{S}_T$ is the union of the read and write sets of the transaction program $\mathcal{W} \mathcal{S}_T = \{\mathcal{R} \mathcal{E} \mathcal{A} \mathcal{D}_T \cup \mathcal{W} \mathcal{R} \mathcal{I} \mathcal{T} \mathcal{E}_T\}$. A transaction instance $T_j$ (or simply transaction) of a transaction program executes at a given node $n$ with a minimal inter arrival time, expressed as a maximal frequency $f_j$. We define such transaction instance by a tuple $T_j = \langle f_j, n, T \rangle$, and $\text{node}(T_j) = n$.

2.4 Virtual Full Replication

The concept of Virtual Full Replication (ViFuR) (Andler et al. 1996, Matthiason & Andler 2003) has been introduced in DeeDS to ensure that all used data objects are available at the local node, and to reduce the resource usage compared to full replication. ViFuR has the advantages of full replication, such as transaction timeliness, simplified addressing of communication between nodes, built-in storage and data aggregation, as well as support for
2.5 Scalability and resource usage

fault tolerance and partitioning. Virtual full replication creates a perception of full replication to the database user, such that a database client cannot distinguish a virtually fully replicated database from a fully replicated one. Replication is important for local availability that enables transaction timeliness, and to ensure durable storage for the main memory database. Secondary, replication also improves fault tolerance and reliability. By replicating only the objects that are currently in use at a database node, the database scales with the actual required degree of data object replication, rather than with the number of nodes (Definition 2.2) (Mathiason & Andler 2003).

Definition 2.2. : In a system with Virtual Full Replication (ViFuR) there exists a local replica of an object used by each transaction that reads or writes the database objects at a node, such that \( \forall o \in O, \forall T(o \in \{WS_T\} \rightarrow \exists r \in R(r = R(o, node(T)))) \)

This thesis argues that a fully replicated distributed real-time main memory database can be made scalable by using virtual full replication for effective resource management, and that the degree of scalability achieved can be quantified. Different scale factors influence resource usage differently, and for this reason an evaluation needs to vary scale factors individually to properly evaluate scalability.

2.5 Scalability and resource usage

In this thesis, we use scalability as the ability to augment the scale of a system with appropriate resources needed for its operation. We consider scalability to be an issue about requiring (or simply ‘using’) less resources for operation than are provided, at an increasing system scale and for a certain system scale of interest. An example is a distributed system of computing nodes where more nodes are added. With more nodes in the system, more users may use the system. Also, with more nodes, more resources are added to the system, for example, in terms of processing units and storage.

Scalability is achieved when a system parameter \( p \) (called the scale factor) can be increased while the consumed resources as a function of the scale factor do not exceed the resources that are available. A system is scalable if the growth function for a required amount of resources,


\[ g(p) = \text{required}(p), \text{ does not exceed the function for the available amount of resources, } f(p) = \text{available}(p), \text{ when the system is scaled up for some scale factor, and where the system continues to provide service at the same level of quality. Both resource usage and resource availability follow a function of the scale factor at an increasing scale, and the upper bound for } g(p), \text{ must not exceed the function of available resources, } f(p), \text{ for a range of } p, \text{ from } p_t \text{ to } p_l \text{ (The Scalability Condition, formula 2.1).} \]

\[ \forall (p \geq p_t, p \leq p_l), g(p) \leq f(p) \quad (2.1) \]

Thus, for an evaluation of scalability for a certain system, the specific scale factors and the specific resources of interest must be expressed. A distributed database may be evaluated using the number of nodes as the scale factor, and with bandwidth and storage usage as the resource concerned. For linear scalability, it is the growth of the functions \( g(p) \) and \( f(p) \) that determines the scalability. For scalability in an range \( p_t \) to \( p_l \), the Scalability Condition must hold for every \( p \). Consider the following example of growth of \( g(p) \) and \( f(p) \): Increasing the number of nodes in a distributed database linearly, while the bandwidth usage grows exponentially, the system certainly does not scale. However, if bandwidth usage is constant for a linearly increasing number of nodes, the system scales.

Scalability concepts are well developed and related metrics for scalability are available in a few research areas, namely those of parallel computing systems (Zirbas, Reble & vanKooten 1989) (Nussbaum & Agarwal 1991) in particular for resource management (Mitra, Maheswaran & Ali 2005), and shared virtual memory (Sun & Zhu 1995), the design of system architectures for distributed systems (Burness, Titmuss, Lebre, Brown & Brookland 1999), and for network resource management (Allison, Harrington, Huang & Livesey 1996). Burness et al. (1999) argue that using a single metric is an oversimplification, since an architecture may be limited by several resources used at an increasing scale of some scale factor. Multiple and relevant metrics related to the usage of distinct critical resources may be more useful as a metric of scalability for a specific application type, for example, resources such as bandwidth, storage and processing time.
2.5 Scalability and resource usage

Frölund and Garg define a list of generic terms for scalability analysis in distributed application design (Frölund & Garg 1998):

- **Scalability**: A distributed (software) design \( D \), is scalable if its performance model predicts that there are possible deployment and implementation configurations of \( D \) that would meet the Quality of Service (QoS) expected by the end user, within the scalability tolerance, over a range of scale factor variations.

- **Scale factor**: A variable that captures a dimension of the input vector that defines the usage of a system.

- **Scalability point**: A specific setting of the Scale factor, important to the user, where resource demands suddenly increase more than expected when extrapolating from smaller scale factors. It represents a threshold of the need for resources, where extra resources are needed.

- **Scalability tolerance**: The permitted variation on QoS that specifies the allowed degradation in QoS with an increase of the scale factor.

- **Scalability limit**: The upper bound on the Scale factor of interest to analyze, with respect to the application intended, and where the system is scalable.

- **Scaling enablers**: Entities of design, implementation or deployment that can be changed to enable scalability of the design.

From this work, we learn that there may be an upper bound on the scale factor for the scalability analysis of a specific system. The resource cost for scalability at very high scale factors may not even be of interest to the application, since the application may not reach such high scale factors.

For a system with linear scalability the Scalability Condition must be valid for all \( p \), but for other systems, scalability may be related to only certain values of \( p \). In this thesis, we consider scalability to be related to a range of interest for the scale factor \( n \) use (Figure 2.1). An upper scalability limit, \( p_l \), may exist, as an limit of the scale factor where consumed resources may exceed available resources for higher scale factors. Likewise, a lower scalability threshold, \( p_t \), may exist where a lower scale factor consume
more resources than are available. The scalability threshold represents an initial cost, or an overhead of the demand for resources, and is outside of the scale factor settings of interest of the application analyzed for scalability. The scalability threshold and the scalability limit define the range of a scale factor where a system to be analyzed is scalable (the range where all \( g(p) \leq f(p) \)).

![Graph showing scalability threshold and limit](image)

**Figure 2.1: Scalability**

A general intuition of scalability is that a system must scale for all scale factors. In this thesis, we elaborate general scalability to also include the scalability limit and threshold, and to distinguish four types of scalability:

- *Linear-scalable* for a range of \( p: [0, \infty) \)
- *Upper range-scalable* for a range of \( p: [p_l, \infty) \)
- *Lower range-scalable* for a range of \( p: [0, p_l] \)
- *Closed range-scalable* for a range of \( p: [p_l, p_l] \)
Range-scalable systems are scalable within a range of interest for the problem or the application. Different approaches for scalability can be compared within that range, while scalability outside of the range scalability is not considered. Throughout this thesis, the type of scalability considered in different sections is not explicit. In general, we search for linear-scalable solutions, but when evaluating typical applications, we typically use a range of interest for an application using a distributed real-time databases.

2.6 Durability and Incremental diskless recovery

As a part of the ACID (Atomicity, Consistency, Isolation, and Durability) properties of a database (Gray & Reuter 1993), storage must be durable. The result from committed transactions need to be durable, and since main-memory databases are typically not durable over a power-loss, there is a need for complementary, durable storage to preserve data in case of failures, such as power or node failures. Typically, main-memory databases use an archive database on disk for durable storage (Eich 1986). However, distributed and replicated databases can enable durable storage by ensuring multiple replicas of the same data. Replicated data can still be stored in main memory but at different physical nodes. Such storage survives node failures up to a certain number of nodes failing, and as long as there is a single replica left, the data is preserved. After such a failure, the system cannot tolerate any further failures until the fault tolerance level is re-established again. In order to re-establish the level of fault tolerance, there is a need for schemes to detect the faulty state, in order to initiate and recover the fault tolerance level again. This includes setting up the missing replicas again, preferably during execution time, so that the database can continue to run without the need to stop the database application and disrupt the service during the recovery. Typically, databases that store data on volatile media use recovery approaches that include logs and checkpoints. Checkpoints transfer committed data to durable storage, while logs save the committed data in the time between checkpoints. At recovery from the archive database, the log is applied on the checkpoint. Checkpointing typically locks the entire database during its progress, disallowing data access for a moment.
**Background**

Fuzzy checkpointing can be used to continuously replicating committed updates to durable media while the database is in use. In incremental diskless recovery (Leifsson 1999, Andler, Leifsson & Mellin 1999), a buddy node is a selected peer node, which is updated by fuzzy checkpointing and that keeps a backup replica. Under the fault tolerance assumption that only one of the two nodes will fail at a time, and that the communication link between these two nodes does not fail, storage is durable. A failed node can recover data from its buddy node, into a consistent replica, without the need for stopping the operation of the buddy node. Updates received at the buddy node are logged, and the log is sent to the recovering node after that the checkpoint has been sent. Once the entire log has been transferred, including all updates appearing at the buddy node during the transfer of the log, the recovery target becomes fully consistent with the recovery source.

In this thesis, we use incremental diskless recovery to setup new replicas of objects and groups of objects (compared to setting up an entire database), such that we use database objects as the smallest entity locked during checkpointing (compared to using memory pages) (Section 6.2.4).

### 2.7 Simulation-based studies of systems

Simulation is often used to study complex phenomena that are unfeasible or difficult to control as a real system. A simulation study can be used to simulate large-scale systems that otherwise could be unfeasible or impractical to realize. However, simulations require careful modeling of the real-world phenomena, so that the simulation is valid for the phenomena to study.

For feasibility, a simulation study covers only essential parts of a real phenomenon to be examined, and in this thesis we choose to model essential components of a database system in detail, while other parts use simplified models. It is a time-consuming task to develop a generic simulator of a specific system from the ground up, thus using an existing simulator is more efficient. Such a simulator needs to accurately model essential features of the system to study. A simulation study where researchers implement the phenomena to be studied, rather than combining packages in a simulation framework creates a better understanding of the phenomena. Details of the
2.7 Simulation-based studies of systems

Simulation-based studies of systems can be controlled and alternative approaches examined. The simulation model needs to cover the essential features to be studied in detail, while some modeling may be simplified. An example of simplified modeling is the actual storage of data in a database simulator. If the simulation study aims at resource usage for replication protocols, the data itself need not be stored, but instead a representation of the actual database storage space using a size value is enough for the evaluation. Due to this simplification, less resources are used at the simulation computer, while replication can be studied. Instead of simulating every processing step at a low lever, selected higher level abstractions may enable a simulation study using hundreds of nodes instead of just a few nodes.

Motivation for a simulation study

In this thesis, we use a simulator to model a real distributed database system. With a carefully designed simulation model, we have the opportunity of reaching an understanding of the resource usage in large-scale distributed databases that use Virtual Full Replication. Our motivations for using a simulation study are:

1. Large scale experiments with the prototype database system are unfeasible, due to the amount of hardware, as well as the size and management of an installation that is required.

2. Executing large-scale experiments with the actual system would require hardware and software engineering efforts significantly higher than with a simulation study, since a database prototype is expected to result in a working database implementation that can be studied. With a simulation, the implementation can be focused on, for example, the replication processing in isolation.

3. An analysis of an actual system could be very complex. Development of a simulation model provides detailed understanding of parts of the system that are modeled carefully, while other parts are simplified. This gives the modeler a chance to focus on certain features connected to the research of interest, while omitting irrelevant details out. While
the actual system is ultimately the best model that can be studied, its development is very costly compared to a simulator.

4. With a simulator, all execution parameters can be controlled, and the simulation experiment can be replicated in a predictable way. However, with a real-world system, many random variables cannot be controlled, and the replication of experiments needs to be evaluated with more samples, using more statistics processing. To test and verify the simulator implementation, we can use the same seed value for repeated executions. Such controlled verification increases the confidence in the simulation model and its implementation.

Validation of the simulation model

Simulating a phenomenon of interest involves creating a model of the system to be studied. Modeling inherently means making a simplification of an actual system, and gives a model that is only valid under a set of known assumptions. Basic requirements for simulations of computer systems can be found in literature (Jain 1991, Banks, Carson & Nelson 1996, Law & Kelton 2000). In order to measure and decide on a simulation model, it is a requirement that the model can be assessed to be correct for the intentions of the study, following the setup of the simulation objectives. For such an assessment, including validation and confidence, the simulation objectives need to be clearly stated. The literature on the validation of simulations is abundant, ranging from high level approaches of establishing taxonomies for simulations in general, to detailed work on simulations of parallel computing. Several authors stress the significance for the validation of simulation models of large-scale systems, in particular for distributed databases (Banks et al. 1996, Sargent 1996, Balci 1998).

A simulation study is useful where the behavior of the real system cannot easily be analyzed. This includes when the input or the model has some stochastic component, or when computation of an analysis is complex. To evaluate such simulation study results, statistics are important. However, not all statistic techniques may properly be applied, since many computer systems give responses that do not have normal distribution (Kleijnen 1999).
The phenomenon to be studied, and the evaluation of a simulation, therefore need to be examined for non-normal distribution behavior.

In work by Shannon (Shannon 1981), it was concluded that a simulation model cannot be a complete representation of the system to be studied, but that a simulation model needs to be reduced to meet particular objectives of the study. Detailed, general purpose simulations tend to be very costly in development as well as processing time. Validation of simulations is said to be the process of determining whether a simulation model accurately represents the system, according to the objectives chosen for the study. Furthermore, validation of simulations is a matter of creating confidence that the simulation model represents the system under the given objectives. Simulation confidence is not a binary value, but is gradually strengthened by tests for validity. In Sargent (1992), concrete tests for validation are presented. These can be used as control questions in an evaluation of the simulation for validity: 

Degenerate tests - How does the model’s behavior change with changed parameters? 
Event validity - How does a sequence of events in the simulation correlate to the events of the real world system? 
Extreme-Condition tests - How does the model react to extreme and unlikely stimuli? 
Face validity - How does the model correspond to expert knowledge about the real system? 
Fixed values - How does the model react to typical values for all combinations of representative input variables? 
Historical data validation - How is historical data about the real system used for the simulation model? 
Internal validity - How does the model react to a series of replication runs for a stochastic model? For high variability, the model can be questioned. 
Sensitivity analysis - How does the effect from changing input parameters influence the output? The same effect should be seen in the real system. The parameters that have the highest effect on the output should be carefully evaluated for accuracy, compared to the real system. 
Predictive validation - The outcome from the simulation forecast and the execution of the real system should correlate. 
Traces - The behavior of execution paths should correlate. 
Turing tests - Expert users of the real system are asked if they can discriminate between outputs from the real system and the simulation.

The accuracy of a model must be sufficient to allow an evaluation the
objectives of the simulation study. A more accurate model than required will result in a simulation that uses excessive resources, and which may not even be useful to run due to long execution times. In order to establish the level of modeling detail, the simulation objectives must be defined. It is unfeasible to model all aspects of the system (absolute isomorphism). The effort should thus be to model at a detailed level and a representation that is appropriate for the objectives. Only reality itself is a complete, valid model.

**Credibility of the simulation model**

A model is validated on the confidence in the model being able to reflect the real world problem in terms of the objectives to be evaluated. For validity, it is more important to have a model with confidence than to have a fully detailed and accurate model. It is therefore important to know how to increase the confidence in the model. Robinson (1997) describes how validation is improved by addressing model confidence: Iterated verification and validation need to be used along with iterative development of the model itself. In addition, the view of the world may not hold for validation. Real world data is often inaccurate and a representation of the world is the result of a certain interpretation of the world. There may not even be a real world situation for comparison. Consider a full scale nuclear war. This is a likely subject for simulation, but it would be very hard to compare to a valid real world situation. Robinson refers to common techniques in the area with which to handle typical problems in validation:  

*Conceptual modeling:* It is important that the modeler acquires a deep understanding of the real world system to be modeled, which requires close collaboration with domain experts. In this way, the modeler can understand alternative interpretations from which objectives and a conceptual model can be developed. The domain experts need to validate the conceptual model with a review of the modeling objectives and the modeling approach.  

*Data validation:* Real world data must be validated for accuracy by exploring and valuating the data source to determine data reliability, completeness and consistency.  

*White-box validation:* Inspection of the simulation implementation improves the correctness of the model implementation and compliance with the con-
ceptual model. Control of events, flow and logic is central and code reviews, execution trace analysis and output analysis are used for this. 

**Black-box validation:** Inspection of the overall behavior of the model also includes correlation analysis, comparison with the behavior of a real world system. Comparison can also be done with another model or simulation of the same or similar problem, but simulation objectives and problem differences must be considered.

For a credible model, there is an agreement that the model reflects the real system appropriately for the experiment (Fossett, Harrison, Weintrob & Gass 1991). To establish credibility, there must be an agreement on the assumptions of the model, and proofs of validation and verification are needed. There is a risk that an agreed model, that is regarded as credible, is still not valid due to improper validation. To further improve credibility, the model can be accredited. Accreditation is the formal acceptance from some authority that the model and the simulation are approved to be used for a specific experiment or usage (Balci 1997).
Chapter 3

Problem Definition

The most valuable of all talents is that of never using two words when one will do.
- Thomas Jefferson

This chapter states the thesis problem of scalability in distributed real-time databases that have full replication, and which require timely transactions. We present the thesis statement, and show how the thesis problem is divided into individual subproblems that need to be solved, in order to solve the thesis problem.

3.1 Problem introduction

Centralized, real-time databases are limited by the resources available at a central computing node, while distributed systems scale better when resources are added with the increase in the number of nodes. Scalability is an increasingly important issue for distributed real-time databases, since more and increasingly larger distributed real-time database systems are being built. Scalability-enabling approaches are required to enable such systems. The distribution and replication of data are important scalability enablers that keep operations as local as possible, without the need to depend on a central
node or even a group of close nodes for its operation. Localized operations need data to be placed locally. With a fully replicated database, all data is available at each node, such that all accesses to the database can be done at each node locally. With multiple replicas of a data object, consistency management between replicas is needed. A scheme with both full replication and consistency management enables concurrent updates to replicas of the same data at different nodes. Transactions in fully replicated real-time databases can access arbitrarily selected database objects at any node, but such flexibility has an exponential cost in resource usage for large systems. Full replication requires that all nodes receive every update of a replicated data object at any other node. With increasing demands for larger distributed real-time databases, there is a need for scalability-enabling approaches that have the localized operation advantage of full replication. Such approaches need to scale, in a way that does not overuse resources as the system grows.

### 3.1.1 Scalability in fully replicated databases

A fully replicated database with detached replication of updates (such as DeeDS, Section 2.2) scales better than a fully replicated database that uses a distributed agreement during the transaction for an update with full consistency. With detached replication, updates can be allowed concurrently at all nodes, independently of a distributed agreement between replicas. However, another scalability problem caused by full replication remains, since all updates in a fully replicated database need to be sent to all other nodes. Consider a database with $n$ nodes and $o$ data objects. An update to each data object must be replicated to $n - 1$ nodes. In a worst case situation where all data objects are updated, in principle $O(on)$ update messages must be sent, regardless of whether the object replicas ever will be used at all the nodes. Also, full replication requires that all $o$ data objects are stored at all $n$ nodes and, for conflicting concurrent updates, the inconsistencies would need to be resolved. For some applications, the size of the database may increase with a degree of the number of added nodes, such that $o = f(n)$, implying that $o = O(n)$. With such behavior, the number of required messages would increase at $O(o^2 n)$ with the number of nodes.

For many database systems, it can be assumed that only a fraction of
3.1 Problem introduction

the database is actually used locally at a node, which can be motivated by the typical locality behavior of accesses in distributed databases. Further, a small set of objects are likely to be accessed more often than other objects, effectively forming a hot-spot of the logical database. When using knowledge of the actual data needs and recognizing differences in the properties of the data, means there is an opportunity to maintain the same perceived availability of data for the application as with full replication. Such an approach reduces the flexibility of allowing arbitrary accesses at any node. For hard real-time systems, access patterns and resource requirements are usually known a priori, and the flexibility gained by full replication is therefore less critical for such systems. Soft real-time systems can typically not be fully pre-specified, and arbitrary accesses need to be allowed. When considering the actual need for the data, the number of replicas can be bounded by the actual data requirement of the application using the database.

With replicas of data objects at only a few nodes, committed updates need only be replicated to a subset of all the nodes; the ones that currently host replicas of the updated data object. With a bound on the degree of replication, $k$, where $k << n$, and $n$ is the number of nodes, the required number of update messages is expected to scale with the degree of replication rather than with the number of nodes, since updates are sent to $k - 1$ nodes instead of $n$ nodes.

Such reduced replication will bound bandwidth requirements for replicating updates, and requires less main-memory storage since not all the nodes will host all the objects. Also, the processing cost of conflict detection and resolution for inconsistent replicas will be lower since fewer nodes host conflicting replicas.

3.1.2 Allocation of data object replicas

One opportunity of reducing resource usage is to find a balance between the number of replicas actually required for acceptable performance, at a cost reasonable for accessing these replicas. The general problem of optimal allocation of replicas in a distributed system, called the File Allocation Problem (FAP), is an NP-complete problem (Casey 1972, Chandy & Hewes 1976). The problem thus needs to be addressed by appropriate
heuristics and algorithms suitable for specific usage. This thesis addresses the allocation of data replicas under the constraint that, for timeliness of execution, every transaction should access data objects only at the local node of execution. Such a constraint disallows data access at remote nodes and calls for the need of information about all the accesses to objects, for each node in the system.

The DeeDS database system prototype is a main-memory resident, fully replicated database that uses detached replication of updates. With detached replication, the network delay for updating replicas is separated from the execution time of the transaction. The combination of main memory residence, full replication and detached replication of updates gives timely transactions in a distributed real-time database, where execution timeliness of transactions depends only on access times to local main-memory. This thesis explores and evaluates the usage of Virtual Full Replication (ViFuR) (Section 2.4) in a distributed real-time database system with eventual consistency, such as DeeDS. The concept of ViFuR was proposed in order to give a database user the perception of a fully replicated database (Andler et al. 1996), as an approach that has the advantages of full replication, while avoiding the excessive use of resources. By using knowledge about data needs, irrelevant replication can be avoided while maintaining user perception of full replication.

### 3.2 Problem statement

In partially replicated distributed databases in general, the clients of the database need to have knowledge of replica locations in order to access data objects. For timely access in distributed real-time databases, all the resources involved in accesses must be bounded to achieve timeliness. Many approaches achieve it by using advanced control of distributed transactions and load balancing between nodes. In addition, concurrent access of replicas of objects in a distributed database escalates the complexity, since the database user must also model the behavior of the resource usage of other accesses. There is usually little knowledge about the accesses requirements of other clients using the same database. Therefore these requirements need
3.2 Problem statement

to be modeled as a worst case usage of resources.

With full replication, the complexity of managing accesses can be reduced, while achieving timeliness of execution. The data client becomes location independent by replication transparency, and can use the entire database without explicitly managing resources. With eventual consistency, concurrent updates to replicas of the same object are independent, and all operations on database objects can be timely performed at the local node. However, such a system does not scale.

The problem to solve in thesis is two-fold:

1. **Scalability**: How to make a distributed real-time database with timely transaction execution scalable.

2. **Flexibility**: How to allow timely transactions that cannot be pre-specified prior to system execution, in a scalable distributed real-time database.

Based on this problem definition our thesis statement is:

*A full elaboration of Virtual Full Replication by segmentation is an approach that enables scalability in distributed real-time systems with timely transaction execution, such that both pre-specified and on-demand data needs are satisfied, and where transaction timeliness and scalability is adaptively maintained as data needs evolve.*

In this thesis we chose to pursue ViFuR as a scalability enabler primarily for distributed real-time databases that have eventual consistency. ViFuR may also improve performance for distributed databases in general, since the number of remote operations are reduced dramatically. ViFuR manages replication by using knowledge about data needs, to avoid unnecessary replicas. In order to maintain scalability, a ViFuR approach must also manage replication to meet concurrent requests for resources. ViFuR is explored and elaborated as a resource management approach to reduce the amount of resources required. A database with ViFuR is capable of adapting the allo-
cation of resources for both static and changing data needs, while preserving the timeliness and scalability of the application.

3.3 Problem decomposition

The two-folded main problem of the thesis is pursued as the following subproblems.

1. Scalability-related subproblems: Full replication as an approach to transaction timeliness introduces the problem of scalability, caused by exponential resource usage at increased scale factors. This subproblem is addressed in Chapters 4 and 5.

   (a) The initial proposal for using ViFuR (Andler et al. 1996) needs to be pursued. How is a generic approach to manage a virtually fully replicated database built, and how is a scalable approach for replica management designed? An approach also needs to allow the use of multiple objects properties to support replication, consistency and also other properties.

   (b) How is a scalable approach for replica management formally defined?

   (c) How can a management approach of replica allocation be based on known requirements of data needs? How can multiple requirements on data needs be supported? How can conflicting requirements be found and resolved? How can scalable algorithms and data structures for management of such an approach be built in an actual implementation.

   (d) How should a generic architecture be designed, with algorithms and replication management structures? Having such an architecture, the ViFuR approach can be evaluated for the usage of key resources, for an evaluation of the scalability of the approach, and for conditions of interest.

2. Flexibility-related subproblems: Using ViFuR in a database and the pre-specification of data results in a problem of lost flexibility caused
3.3 Problem decomposition

by the reduced number of replicas available compared to full replication. The flexibility of full replication is required when accesses cannot be pre-specified prior to execution. For many applications, it is hard or impossible to pre-specify data needs. Typical distributed real-time applications have mode changes that require different data to be available at different times. In addition, the actual data needs may arise from unpredictable events in the environment. Furthermore, for scalability, flexibility and availability over time, it is necessary to manage resources to avoid the use of resources for no purpose, such as when replicas are not longer used due to mode changes. This sub problem is addressed in Chapters 6 and 7.

(a) How can an architecture support the detection of evolving data needs for adaptive replica allocation management?

(b) How does a distributed algorithm work that implements detection of the changes of data needs and allows concurrent adaptation of the replication schema? Such adaptation changes must result in a consistent allocation of objects at all nodes affected by adaptation, and provide consistent information about the current allocations available at each local node.

(c) How can we solve the problem of maintaining scalable global scope information for finding current allocations of objects? Such global information is required when establishing new replicas at adaptation.

(d) How can an adaptive approach automatically any discontinued need of replicas, as well as remove excessive ones. Such detection required generic removal processing, which can also be extended to make use of knowledge about application-specific access patterns.

(e) What is the scalability improvement, compared to alternative replication approaches, when using the approach in a large-scale controlled environment setting, such as a simulator? An evaluation is needed for chosen experiment variables with well-defined settings for typical usage.
With solutions for these subproblems, the approach also needs to be evaluated in typical scenarios employing application-related usage constraints, such as a Wireless Sensor Network (WSN). Scalability is evaluated in terms of resource usage in bandwidth, storage, and adaptation processing delays within the constraints given by such a typical usage scenario. Further, we also seek improvements that can be made by utilizing the properties of such a real scenario, for example, application topology and locations of nodes. Using real scenario properties to improve the limitations of the generic approach are of particular interest.

We have chosen to pursue the exploration of virtual full replication in three distinctive steps. The first step is to develop an approach for scalable data management by the segmentation of the fully replicated database (Chapter 4). Employing a generic approach, we present static segmentation, using a method for the management of a statically segmented database and a model architecture (Chapter 5). Subproblem 1 is addressed in this step. The second step elaborates static segmentation by allowing changes to segments such that all objects become logically available at each node. This re-establishes the flexibility of full replication that is lost by segmentation, and re-enables arbitrary accesses of data at any node of the database. We also present an algorithm for the removal of replicas no longer in use (Chapter 6). This step addresses subproblem 2. The third step is to apply constraints from a typical usage scenario for an evaluation of our overall approach for ViFuR (Chapter 7).

3.4 Assumptions

This thesis addresses the scalability problem of a distributed real-time database. We exclude the consistency management of replicas. With a consistency management scheme in place, independent updates are allowed, and conflicting updates are detected and can be resolved. Further, the problem of consistency management with network partitioning is excluded. With a consistency management scheme assumed, partitioned operation, reconnection, and reconciliation of replicas is handled.

The ViFuR approach presented in this thesis is developed for a distribu-
3.5 Research methodology

The methodology of this thesis is both exploratory and problem-solving (Robson 2002). We pursue and elaborate the concept of ViFuR as a scalability approach for distributed real-time main memory databases. We also develop the concept into a concrete set of schemes, using a detailed system model and software experimentation for the evaluation of the approach and the schemes. Four methods are used:

1. **Literature survey.** This method is used in the initial stages of the work both to understand what need of scalability implies for the database systems intended, and to develop an understanding of the approaches already used in the literature. Furthermore, approaches for similar problems in other domains are searched for. By using this method in the initial stage we find, for instance, that the complexity of timely and strong consistency models make weak consistency schemes suitable for large-scale systems. In addition, we learn about existing alternative approaches for both the selective and adaptive replication of updates in distributed systems with multiple replicas. This method formulates the problem and structures relevant work.

2. **Development of a base approach.** Based on the survey findings, we formulate a base approach for scalable resource management for dis-
tributed real-time main memory databases. In order to enable evaluation we define scalability and evaluate it for the base approach in terms of usage of critical resources. This is an evolutionary method for developing a prototype for the approach.

3. **Refinement of the base approach and evaluation, using detailed models and controlled experiments.** With requirements for data access that cannot be pre-specified, a simple analysis model is insufficient. However, we develop a detailed model as a tool to evaluate the use of the approach in a large-scale setting. The distributed real-time main memory database model is highly detailed, such that actual behavior can be studied and examined, and also further developed. A set of experiments with carefully chosen variable settings are conducted and the results are examined. This method develops the prototype concept into a full model of the concrete approach, which is evaluated using the model.

4. **Generalization of the approach and extensibility test.** We apply the constraints of a realistic large-scale scenario, to enable us to explore the extensibility of the adaptive base approach. Such an extension provides a solution to the problem for the specific scenario, but also reveals how extensible the base approach is. This method acts as a validation of the usability of the approach, and as an applicability test of the approach for a specific scenario.
Chapter 4

A Segmentation Approach for Virtual Full Replication

Divide and Conquer
- Julius Caesar

This chapter elaborates on the concept of Virtual Full Replication (Vi-FuR) for usage in distributed real-time databases. We present how properties of data objects of the database can be used for the segmentation of the database, to allow individual degrees of replication for subsets of objects in the distributed database. In addition, we show how relations between object properties can be used for the segmentation of a database. The usage for a few chosen useful data properties and their relations, as well as how such relations can be applied to reduce the number of segments are also described. Finally, we introduce and use the notion of multiple segmentations.

4.1 A perception of full replication

In a fully replicated database, all the data objects in it are available at the local node of any transaction, allowing arbitrary data access to any of the database objects at the local node. Full replication uses excessive resources and does not scale (Mathiason & Andler 2003). Virtual Full Replication
(Definition 2.2) makes all database objects *logically available* at the local node, and only the used database objects *physically available* at the local node while needed. Logical replicas have no physical manifestation, but a mere representation that allows availability by reference, rather than by instance. This creates a perception of full replication to the database user, such that the database can be used in the same way as with full replication.

With ViFuR, resource usage is proportional to the amount of concurrent need for database object replicas, rather than to the size of the database or the number of nodes. The database management system (DBMS) conceals the management of allocations and reallocation mechanisms, such that the implementation of ViFuR is hidden behind the database transactional interface used by the database user. ViFuR manages knowledge about where all the used replicas of database objects are located. In a typical database application there is a higher demand for certain objects, such as they have both temporal and spatial preference above other objects. Such data is often referred to as *hot data*, and is typically modeled with *b-c accesses* (Tay, Goodman & Suri 1985). B-c accesses model hot-spot behavior by accessing a fraction, \( c \), of the database objects through a share, \( b \), of all the accesses of transactions. As a common model of hot-spot accesses, the values for \( b \) and \( c \) are often set at 80\% and 20\% respectively, so that 80\% of the accesses goes to 20\% of the data (Wu 1993). In our work, we assume b-c accesses, such that only a few nodes and replicas are concurrently accessed for the same data object. With such typical data usage, it seems to be a large resource waste to allocate objects to all nodes in a distributed database.

With the assumption that only a few replicas of database objects are concurrently needed for actual usage of data, there is a potential for a dramatic reduction of resource use compared to full replication. This chapter presents a generic approach for ViFuR applying database segmentation that can manage the currently needed data so that the replication of updates, adaptation to new data needs, and consistency management, are concealed to the database users. This is achieved so that the users need to have no knowledge about the location of the replicas of the data objects. The application can be written as if all the database objects are locally available, which simplifies the application since update synchronization is removed from it. Also, local
availability of data is essential for the transactional timeliness of real-time
databases (Section 2.2). For a database with ViFuR and a scheme for event-
tual consistency, there is no need to access remote data at all. There is also
no need for the database user to coordinate and serialize updates between
multiple replicas. Transactions execute timely at the local node, indepen-
dently of unpredictable network delays and time-consuming synchroniza-
tion mechanisms, or without the need of a system designed for a known
number of nodes (such as a majority of the nodes) for successful updates
of the database (Gifford 1979). This and the two following chapters elab-
orate on the generic approach. A method for scalable static segmentation
in multiple properties is presented in Chapter 5, while the scalable usage
of ViFuR in a system where data needs change over time is described in
Chapter 6. Finally, the usage of ViFuR in a large-scale scenario of wireless
sensor network nodes is covered in Chapter 7.

4.2 Database segmentation

As an approach for ViFuR, we use segmentation of the database as a method
of grouping related objects for replication and resource management. We let
each segment manage properties of data objects. One important property is
the allocation of a data object to a node, and this property gives each seg-
ment an individual degree of replication. This section elaborates on database
segmentation as we use it for ViFuR, for management of groups of objects
that share the same properties.

4.2.1 Object properties and segments

A database object has implicit properties, such as its memory size, or the
media type used for storing it. There may also be properties imposed on
the object by the environment, such as the update interval or data freshness,
or the validity interval of updates. Further, the database management sys-
tem may also impose properties that limits what data object properties can
be satisfied. Such constraints originate from the database implementation
or the architecture, such as the consistency management, replication meth-
ods, and the storage available. They may also originate from how resources are used by the database, such as bandwidth of communication links or the memory available for storage. In addition, the database application may require that certain data properties hold, which may be in conflict with the database implementation or architecture. We present a segmentation framework that enables segmentation on a database for multiple data properties. In this framework we use a chosen set of useful properties for a proof-of-concept segmentation of a real-time distributed database.

4.2.2 Segmentation

In this thesis, a segment of a database is a set of objects that share a certain set of properties, such as the nodes where data is allocated, or what consistency model objects use. Segments can be uniformly managed across the nodes of a distributed database. For instance, grouping data objects on "data replica allocation" will result in segments that share sets of the nodes in which objects in the segments are allocated. Such a grouping of objects enables the uniform processing of groups of objects; the replication of updates can be sent in blocks to entire segments in order to reduce replication overhead, the recovery of segments and the prioritization of segments in recovery. Separate segments can also use separate storage media, as well as individual consistency models. The management of segments needs some extra storage for properties of the objects in the segment, and these properties are stored once for each segment, not for each object. In order to simplify the presentation of segmentation as we use it, we first introduce only the property of "allocation" of data objects to nodes. Later, this is generalized into the usage of multiple properties, to elaborate its problems and approaches. The thesis focuses on static segmentation in chapter 5, while subsequent chapters support adaptive segmentation as well.

Segments are mutually exclusive sets in a segmentation of a database, such that a database object belongs to one segment only for that segmentation. Further, a database can be segmented by different segmentations, which represent orthogonal sets of properties used as the basis for the segmentation. For each segmentation, an object belongs to one segment only. Segments are units for management of data objects in order to reduce the
overhead needed to manage properties and property settings. Also, different
segmentations are used by the DBMS for different purposes. For instance,
to limit the replication of updates to nodes where data objects reside only,
we need to include object allocation in its segmentation, to send updates to
the replicas in $\mathcal{R}$ (Section 2.3). Definitions 4.1, 4.2, 4.3, 4.4, 4.5 and 4.6
give definitions that we use to describe database segmentation used in the
thesis.

The formalization of segmentation below makes use of the formaliza-
tion of a database introduced in Section 2.3. There, we define and relate
database, database node, data objects and data object replicas. In the same
section, we also define transaction programs and their read and write sets.
Here, we start with a definition of data object properties (Definition 4.1), on
the base of which segments are built. Each data object property have object
property settings (Definition 4.2), either mutual or non-mutual. A mutu-
al setting excludes other settings of the same property, while non-mutual
settings can be combined in a setting. A data object description (Defini-
tion 4.3) contains one or several properties that have the same setting, and
which captures the properties of an object. Objects with identical data object
descriptions belong to the same segment (Definition 4.4), and share the same
property settings for all properties used in the database. A segmentation
(Definition 4.5) on an object description may contain all properties used,
but may also contain a subset thereof. Still, all the objects in a segment for
a particular segmentation share the same object description (they all have
the same object property settings). Finally, the segmentation set (Definition
4.6) is the set of all the segmentations done for the database.

**Definition 4.1.** A data object property, $P_i$, is used to describe a data object,
o$_i$, and is expressed by a tuple $< a, A >$, where $a = \{P_1^i, P_2^i, \ldots\}$ expresses
the alternative allowed settings of a property to describe o$_i$, and where $A$ is
a constraint descriptor for the settings of $P_i$. $\mathcal{P}$ is the set of all the properties
in use for a database D, such that $\mathcal{P} = \{P_1, P_2, \ldots\}$.

**Definition 4.2.** A data object property setting, $p_i$, is an assignment of a set-
ing for a certain property, $P_i$, of a data object, where $p_i \in a$. The cardinality
of a property, $| P_i |$, is the number of alternative settings of the values for a
property $P_i$, with respect to the constraint descriptor $A$. Settings $P_1^i$ and $P_2^i$
are mutually exclusive if \( A = \text{mutual} \), such that \( p_i = \{P_i^j \xor P_i^k\} \), or \( P_i^j \) and \( P_i^k \) are not mutually exclusive if \( A = \text{nonmutual} \), such that \( p_i = \{P_i^j \) and \( P_i^k\} \).

**Definition 4.3.** A data object description is a tuple \(<o_i, p>\), where \( p = \{p_1, p_2, \ldots\} \) is a set of property settings, \( P = \{P_1, P_2, \ldots\} \), to describe an object \( o_i \).

**Definition 4.4.** A segment, \( S \), is a tuple \(<O, p>\), where \( O = \{o_1, o_2, \ldots\} \) and \( p = \{p_1, p_2, \ldots\} \), such that objects in a segment share the same object description. A segment, \( S \), contains objects in a logical database, \( D \), such that each object is assigned to one and only one segment, \( \forall(o_i, o_j, S_k, S_l), o_i \in S_k \land o_j \in S_l \rightarrow S_k \neq S_l \).

**Definition 4.5.** A segmentation, \( \sigma \), is a tuple \(<S, P>\), where \( S = \{S_1, S_2, \ldots\} \) and \( P = \{P_1, P_2, \ldots\} \). For each segmentation there is a unique \( P \), such that \( \forall(P_i, P_j, \sigma_k, \sigma_l), P_i \in \sigma_k \land P_j \in \sigma_l \rightarrow \sigma_k \neq \sigma_l \).

**Definition 4.6.** The segmentation set, \( \Sigma \), for a database \( D \), contains all segmentations \( \Sigma = \{\sigma_1, \sigma_2, \ldots\} \) in use for a database.

### 4.2.3 A segmentation example

A central property used by ViFuR for segmentation is the property of allocation of an object. Consider a segmentation for the required allocations of objects to nodes. The known accesses come from the list of transactions (or rather transaction programs, which are transactions not yet instantiated at the execution nodes, Section 2.3). The working set (the objects used by the transaction) for all transactions at each node can be compiled into an allocation schema for replicas at nodes of required accesses. Segmenting a database on such a pre-specified set of known data references is straightforward. An allocation schema for the database objects can be setup for optimal allocation by a database designer. An allocation schema can also be extracted automatically by a segmentation algorithm using the list of accesses of known transactions at each node.

Consider the following example of segmentation on known accesses. Assume a database \( \langle\{o_1, \ldots, o_6\}, \mathcal{R}, \{N_1, \ldots, N_5\}\rangle \), where seven transactions
4.2 Database segmentation

$T_1, \ldots, T_7$ may execute at nodes $N_1, \ldots, N_5$. Each transaction has a read set (prefixed by $r$), a write set (prefixed by $w$) and an execution node. For this example, transaction $T_1, \ldots, T_7$ are specified by the following tuples:

$T_1: \langle r: \{o_1, o_6\}, w: \{o_6\}, N_1 \rangle$, ("$T_1$ reads $o_1$ and $o_6$, writes $o_6$, at node $N_1$")
$T_2: \langle r: \{o_1, o_6\}, w: \{o_6\}, N_4 \rangle$,
$T_3: \langle r: \{o_3\}, w: \{o_5\}, N_3 \rangle$,
$T_4: \langle r: \{o_5\}, w: \{o_3\}, N_2 \rangle$,
$T_5: \langle r: \{o_2\}, w: \{o_2\}, N_2 \rangle$,
$T_6: \langle r: \{o_2\}, w: \{o_2\}, N_5 \rangle$,
$T_7: \langle r: \{o_4\}, N_3 \rangle$.

From these transaction specifications we derive the following four segments to meet the pre-specified requirements of the transactions used:

$s_1 = \langle \{o_4\}, \{N_3\} \rangle$,
$s_2 = \langle \{o_1, o_6\}, \{N_1, N_4\} \rangle$,
$s_3 = \langle \{o_2\}, \{N_2, N_5\} \rangle$,
$s_4 = \langle \{o_3, o_5\}, \{N_2, N_3\} \rangle$.

Such static segmentation for an allocation schema based on known accesses can be derived off-line, once for the entire execution. Static segmentation for ViFuR utilizes the fact that real-time databases typically have canned transactions (Ramamritham 1993, Datta, Mukherjee, Konana, Viguier & Bajaj 1996), which have unchanged read and write sets over multiple instantiations of the same transaction program. All accesses in such transactions are known when the transaction begins, such that canned transactions will use the same working set every time the transaction executes. There are no alternative accesses in canned transactions, which may be the case for transaction programs with alternative execution paths. However, the static segmentation approach can also support transactions with alternative accesses over multiple instantiations. In this situation, the working set to consider for segmentation must include the union of accesses in all alternative paths. Such assignment of replicas will not be optimal, since more
than the optimal number of replicas need to be assigned at every time instant during the execution to support the union of accesses.

4.3 Segmentation on multiple properties

Segmenting a database on a set of known data references is a straightforward problem (as seen in Section 4.2.3), since an allocation schema can be translated directly from the list of known accesses. Such a translation becomes more complicated when multiple object properties need to be considered in a segmentation, since there will be a combinatorial increase in the number of segments that have a unique combination of properties.

In this section we extend basic segmentation, to approach the problem of the combinatorial increase in the number of segments when segmenting a database on multiple properties, and generating multiple segmentations. We elaborate on the usage of multiple properties, and dependencies between properties used for segmentation. We also show how dependencies between properties can be specified as rules and used for validation of the property settings, so that the number of segments and the segmentation overhead can be reduced.

4.3.1 Using multiple properties of database objects

Consider again the example in Section 4.2. In addition, assume the application requires that $o_5$ resides on disk to ensure persistent storage, while all other objects must be stored in the main-memory, to meet real-time requirements. Such segmentation combines the usage of two properties: allocation and storage media. An object’s storage media constraint is a data object property with two alternative property settings which are mutually exclusive: ”storage in main-memory” or ”storage on disk”. If combining the allocation and the storage media properties for a segmentation, segment $s_4$ in the example will need to become two separate ones, in which the object descriptions for the objects in the two segments are different.

$$s_1 = \langle \{o_4\}, \{N_3\}, \{\text{mainmem}\} \rangle,$$
4.3 Segmentation on multiple properties

\[ s_2 = \langle \{o_1, o_6\}, \{N_1, N_4\}, \{\text{mainmem}\} \rangle, \]
\[ s_3 = \langle \{o_2\}, \{N_2, N_5\}, \{\text{mainmem}\} \rangle, \]
\[ s_{4a} = \langle \{o_3\}, \{N_2, N_3\}, \{\text{mainmem}\} \rangle, \]
\[ s_{4b} = \langle \{o_5\}, \{N_2, N_3\}, \{\text{disk}\} \rangle. \]

By this example we see that segments are split if more than one setting is used by a new property added for the segmentation. Also, the set of objects is divided over the resulting segments, such that the segments decrease in size in terms of the number of objects. With an increasing number of properties and with higher property cardinality of the properties used in a segmentation, the segments will be more and smaller, and the database gets a small-granular segmentation. With smaller-granular segmentation, the relative segmentation overhead will increase, while the benefit from segmenting the database will decrease. A worst case scenario is when each object has a unique combination of property settings, such that each object is the only object in its segment. Equation 4.1 expresses such a worst-case number of segments, \( S \), for a set \( P \) of \( P \) properties, where each property has cardinality \( |P_i| \).

\[ S = \Pi_i^P |P_i| \]  

(4.1)

The intention of using segments is to collect and handle similar objects uniformly. It is an advantage to have few segments, to reduce overhead in replication and storage. More objects can thus be handled with block operations and they can share the storage needed to describe the objects in the segment. To achieve large-granular segmentation where more objects share a segment, it is an advantage to use only a few useful properties and low-cardinality properties. Segments will inherently be larger-granular for applications with the characteristics of much cohesive data (many objects share the same properties and property settings). Further, combining segments that have similar property settings gives larger-granular segmentations. One way is to combine segments by explicitly allowing the application designer to define the cohesiveness of data objects in the database application.

All non-mutual properties can have cardinalities higher than 1. A non-mutual property with potentially very high cardinality that may give a large
number of segments, is the property of allocation. With many nodes, there are many ways of combining sets of nodes for the allocation of data objects. However, one of our basic assumptions is a database application only needs a few concurrent replicas simultaneously. This will naturally group objects that are used at the same set of nodes, place them in the same segments, and reduce the actual number of used node combinations. Thus, for typical applications intended, the actual cardinality of allocation will be relatively low in practice.

### 4.3.2 Related object properties

To segment a database, we first collect all the required property settings. In such a collection, there could be invalid or inconsistent property combinations requested, meaning that no valid segmentation can be formed. To resolve inconsistencies, we make use of rules that capture both limitations for combinations of properties and constraints imposed by the execution environment. This section introduces how rules can be used, by giving examples of how to find invalid combinations of properties for single objects. Property dependencies can be used not only to disallow certain property settings, but also to reduce the number of segments, by making use of rules for property settings for the combination of segments. Consider the following two examples of constraints of possible segmentations:

1. **Constraining relations between property settings.**

   As an example of conflicting property settings, consider again the example in Section 4.2, where we look carefully at $o_2$. Assume that $o_3, o_4, o_5$ need immediate consistency between replicas, while other objects use eventual consistency. Further, we assume that $o_2$ and $o_4$ are required to be stored on disk, while other objects are stored in main-memory. For a segmentation that considers all the properties of $P_1 = \text{allocation}$, $P_2 = \text{consistency}$ and $P_3 = \text{storage media}$ ($P = \{P_1, P_2, P_3\}$), we need to verify that these properties can hold together. For this set of properties and settings, there is a conflict for object $o_2$. We cannot ensure a bound on the time for replication, since the object should be stored on disk. In our database model, disk storage pre-
4.3 Segmentation on multiple properties

vents timeliness of the local execution of transactions. This is clearly a limitation of how properties can be combined, which needs to be considered when setting up static segments. A rule for the architecture may be expressed as "bounded replication requires main-memory residence", which can be formulated as the setting of the property of $P_2 = \text{consistency}$ where the setting is consistency="bounded replication" depends on the property of $P_3 = \text{medium}$ with the setting of medium="main-memory". A remedy would be to relax the requirement of disk storage, and provide durability in some other way. With this defined as a rule, the conflict can be detected during segmentation, and the application designer can be informed to take action.

2. Constraining relations between property settings and the execution environment.

Some property settings may not be possible due to constraints of the execution environment. Consider bounded time replication of database object updates. This can only be supported in a database system where 1) data is stored in main-memory, 2) replication supports eventual consistency or some other scheme that avoids global commit, 3) the architecture has a real-time network with a bounded delivery time of messages over the network. In order for the database system to meet these three requirements, we need knowledge about the architecture available, to include in the examination of property settings required. An additional requirement for $o_2$ in example 1 above, which is that updates must be replicated with a bound on the delay of the update to reach other nodes, needs a rule that involves the execution environment. The requirements for object property settings specify $o_2$ to be stored on disk, which do not match the requirements for bounded time replication, since main-memory storage is needed. Moreover, replication must support eventual consistency, and the architecture must support real-time communication. Conflicts related to the architecture need to be resolved as well, before a consistent segmentation can be carried out. In such a segmentation there are no conflicts between application requirements for the object properties, and none between application requirements and the available execution environment.
In order to solve the conflict found for the support of bounded delay replication, a possible remedy may be to change the configuration of the infrastructure, to include a real-time network. An alternative remedy is to disallow bounded time replication of updates, and cause an error in segmentation for the user to handle manually.

After a validation of the property settings required, a segmentation algorithm for multiple properties can be executed for a consistent segmentation after a validation. In Chapter 5, a scalable algorithm for segmentation of multiple properties is provided, and a scalable representation for an implementation of a collection of the objects, properties and property settings used by the segmentation algorithm is also presented.

4.3.3 Forming and using rules for related object properties

In Section 4.2.1, we suggest that properties originate from different sources. Some are requirements from the application while others are constraints of the execution environment. The examples listed in Section 4.3.2 indicate that there needs to be an approach for capturing relations between properties, such that conflicts between property settings, or property settings and the execution environment, can be found and resolved.

A rule uses the syntax \( \text{setting}_1 \rightarrow \text{setting}_2 \), where \( \rightarrow \) means “requires”. Rules use the notation of properties introduced in Section 4.2.2, and the notation is extended here to include \textit{environment properties} of both the database architecture and the execution environment.

\textbf{Definition 4.7.} \textit{The execution environment} is a set \( \mathcal{H} = \{H_1, H_2, \ldots\} \), that holds environment properties and their possible settings, where \( H_j \) is an environment property, \( H_j = \{h_1, h_2, \ldots\} \), considered by the validation of object properties in a segmentation. \( h_i \) is the possible settings for \( H_j \) of an execution environment.

\textbf{Definition 4.8.} \textit{The execution environment settings} describe the currently available execution environment, as an assignment of \( h_j \) from \( H_j \), where \( h_j \in H_j \), such that \( \mathbf{h} = \{h_1, h_2, \ldots\} \) describes the current execution environment.
4.3 Segmentation on multiple properties

In order to prepare the static segmentation on multiple properties that uses rules, the following sequence for validating the list of objects and properties is applied. This validation sequence is a preceding step, necessary for the segmentation algorithm presented in Chapter 5. In the same chapter we also introduce the usage of application specific rules, which are not based on relational constraints between properties as in this chapter, but explicit requirements from the application, that is, a minimum degree of replication for objects. Such rules are evaluated in this preceding step as well.

1. Collect all accesses from known transactions for a pre-specification of accesses needed at different nodes. These define the required allocations.

2. Add property settings from constraints that can be captured, for properties other than required allocations (such as consistency and storage requirements).

3. Separately list properties of the execution environment, from the list of objects and properties.

4. List generic rules to use for validation, and formulate application specific rules for known constraints.

5. Apply rules between properties and property settings on the collection of objects and properties. There are four types of relations that need to be considered:

   (a) Relations between settings of a single property. This can be mutual or non-mutual as expressed in 4.2. This type of property relation is an inherent rule of the property itself and specified once for each property.

   **Example rule:** Each property $P_i$ used for the segmentation of the database is expressed by a tuple $< a, A >$, where $A$ is a constraint descriptor for $P_i$ (Definition 4.1). Property settings are constrained as *mutual* or *non-mutual* (Definition 4.2). There are two generic rules for dependencies of setting of a single property:
A = mutual → \( p_i = \{ P_j^i \text{ xor } P_k^i \} \),
A = nonmutual → \( p_i = \{ P_j^i \text{ and } P_k^i \text{ and } \ldots \} \).

(b) Relations between settings of different properties used for the same object. Rules based on such relations prevent certain settings for a property when combined with certain settings of another property.

**Example rule:** Consider required property settings for object \( o_2 \) in example 1 of Section 4.3.2. Local timeliness of transaction execution is an inherent property of eventual consistency that requires main-memory storage of the object. There is a dependency between the settings of two different object properties. The rule is formulated:
\[
P_k^i = \text{Consistency} \land p_k^i = \text{eventual} \rightarrow \\
P_l^i = \text{Storage} \land p_l^i = \text{mainmemory}
\]

(c) Relations between properties and the available execution environment. This type of rule detects segments with property settings that are not supported by the execution environment.

**Example rule:** Consider example 2 in Section 4.3.2, and assume that object \( o_2 \) needs a bound on the delay for eventual consistency. Also consider an execution environment, \( \mathcal{H} = \{ H_1 = \text{Network}, H_2 = \text{Replication}, H_3 = \text{Consistency} \} \). The object property setting of bounded eventual consistency can only be valid if the object is stored in main-memory, the execution environment supports a real-time network, detached replication, and supports eventual consistency. The rule is formulated:
\[
P_k^i = \text{Consistency} \land p_k^i = \text{boundedeventual} \rightarrow \\
(P_l^i = \text{Storage} \land p_l^i = \text{mainmemory}) \land (h = \{ h_1 = \text{realtime}, h_2 = \text{detached}, h_3 = \text{eventualconsistencymangement} \})
\]

(d) Relations between property settings of two different objects. For objects that have compatible property settings with other objects, the property setting may be changed to the settings of the other object. Such compatibility is directional, and rules are used to find objects that can be joined with other segments, to reduce the
4.3 Segmentation on multiple properties

number of segments.

**Example rule:** Objects required to use the consistency model of eventual consistency can join objects that need bounded eventual consistency. Eventual consistency is directionally compatible with bounded eventual consistency.

\[( o_i \land o_j, i \neq j, p_i^k = \text{eventual}, p_j^k = \text{boundedeventual}) \rightarrow p_i^k = \text{boundedeventual}, \text{ for } P^k = \text{consistency}, \text{ and compatibility rule } \text{eventual} \rightarrow \text{boundedeventual} \]

(directional compatibility).

**Example rule:** Objects with eventual consistency that are stored on disk can be stored in main memory.

\[( o_i \land o_j, i \neq j, p_i^k = \text{disk}, p_j^k = \text{mainmemory}) \rightarrow p_i^k = \text{mainmemory}, \text{ for } P^k = \text{storage}, \text{ and compatibility rule } \text{disk} \rightarrow \text{mainmemory} \]

(directional compatibility).

The two example rules above are setup for mutual properties with mutual settings. For properties with non-mutual settings, the compatibility is somewhat differently defined. Consider the property of allocation.

**Example rule:** For objects allocated to a certain set of nodes, the same object can be allocated for a larger set of nodes.

\[( o_i \land o_j, i \neq j, p_i^k = \text{nodes1}, p_j^k = \text{nodes2} \rightarrow p_i^k = \text{nodes2}, \text{ for } P^k = \text{allocation}, \text{ and compatibility rule } \text{nodes1} \subset \text{nodes2} \]

(subset compatibility).

Rules of type 5d support the manual or automatic joining of segments, by adjusting property settings that result in fewer segments. Using such rules, the segmentation algorithm can indicate where joins are possible, but lets the designer decide for joins, either by manual re-specification or allowing rules to adjust the settings automatically.
4.4 An architecture for segmentation

In a segmented database where segments may have different consistency models, there is a need for an architecture that can support multiple replication methods. As shown in Section 4.3.3, a database user may specify data object properties that need to be supported by a certain execution environment, $\mathcal{H}$, where an execution environment property, $H$, may represent multiple settings. One such $H$ is the set of replication methods available. This section presents a model architecture that supports segmentation, such that different consistency models can be used by different segments, and where the architecture provides multiple concurrently existing replication methods.

Replication for multiple replication methods can be provided in a replication architecture that uses scheduled replication, to prioritize propagation and integration messages locally at each node. This section introduces an extensible architecture with scheduled replication. We see the need for the following four main replication methods, where the architecture must support methods for both full consistency and eventual consistency. The architecture is extensible to support additional replication methods.

1. Immediate Consistency Replication: With full consistency, updates are committed at all replicas at the same time instant. This is a traditional consistency model, which requires a distributed agreement protocol between all replicas. Such an agreement protocol may be a two-phase commit in the group of replicas (Gray 1978) or in a quorum group (Gifford 1979), requiring the updater to communicate with other replicas twice during the update, and in locked steps.

2. Best Effort Replication (also called ASAP Replication): With eventual consistency and best-effort replication, updates are sent out to other replicas as soon as the system resources allow it. Best-effort replication is useful in a system with eventual consistency, where updates are committed at the node of the update, before propagation to other replicas at best-effort. Other replicas receive the update as soon as possible.
3. **Bounded Time Replication**: With eventual consistency and bounded-time replication, updates need to be sent out from the node and integrated at other replicas within a bounded time. Bounded time replication is a type of eventual consistency that allows locally committed updates before propagation. In addition, there is a bound on the replication time.

To support bounded time replication, the system relies on a) Real-time communication that ensures delivery time b) The sum of bandwidth needed from all updates that require bounded time replication at all nodes does not exceed the bandwidth available in a single-segment network c) The propagation and replication mechanisms are deterministic and have bounded processing time.

4. **Deferred Replication**: With eventual consistency and deferred replication, updates are propagated after an unbounded delay, and with lower priority than in best effort replication. Propagation is typically not initiated by the update (push propagation), but rather when there is a need for the update (pull replication).

Deferred replication can be used in an eventually consistent database, which delays even more the integration of updates at all replicas, compared to best effort replication. In addition, replicas need more time to converge into consistency, or they will not converge at all with unbounded delays. Deferred replication is used by systems in which there is no need for updates to be sent before a certain criterion is met. It is a type of on-demand replication that postpones updates until a) A certain time has passed or been reached (Sheth & Rusinciewicz 1990, Xiong, Ramamritham, Stankovic, Towsley & Sivasankaran 2002) b) A certain amount of change of the data has been reached (Gustafsson, Hallqvist & Hansson 2005) c) A certain level of inconsistency has been reached (Yu & Vahdat 2002) and replicas need to be reconciled d) The data user explicitly requests an update of its replica (Sheth & Rusinciewicz 1990). The delay for propagation is typically application dependent.
4.4.1 Model design of a database node

For our problem definition and our approach, we make use of the design and architecture used for the DeeDS database prototype. However, the problem of scalability in eventually consistent distributed real-time databases is applicable to any such database system. In this section, we propose a generic architecture for such database systems, with key components needed for an implementation of virtual full replication in an eventually consistent distributed real-time database. Our focus is on scalability and the support for segmentation.

Each database node in such a distributed database is a peer node, and the key functional component we use can be found in Figure 4.1. Those components are motivated as follows: The Transaction Manager execute transactions from sensor nodes or clients, once the Admission Manager admits the transactions. An Admission Manager is required to reduce the load and initiate contingency actions in case an unexpected number of requests arrive (Hansson 1999). The Segmentation Manager implements ViFuR and manages which replicas to store in the Data Object Store, as well as where and how to replicate updates. The Segmentation Manager maintains segments for the replicas at the local node, and initiates requests for any replicas required from other database nodes. The Segmentation Manager also responds to requests for establishment from other nodes, and acts as a recovery source node. The Logger stores all committed user transaction updates to be propagated by the Propagator to replicas of updated objects at other nodes, according to the segmentation maintained by the Segmentation Manager. Updates received from other nodes are integrated by the Integrator to the Data Object Store, and conflicting updates are found and resolved by the Resolver. All these components, apart from the Admission Manager and the Resolver part of the Integrator component, are simulated in our evaluation of ViFuR in Chapters 6 and 7.

We propose an architecture, where four different replication and consistency models coexist, as listed in the beginning of paragraph 4.4. This generic architecture can be conceptualized as queues (Figure 4.2) with a semaphore for network access, and a scheduling algorithm (Algorithm 4.1, 4.2) which chooses from the queues updates to be sent at the network. The
network is a shared resource that needs to be scheduled according to the consistency classes of the updates to be propagated.

Replication of multiple consistency classes requires support for multiple replication protocols in order to coexist in the same architecture while concurrently using a shared network. Replication methods use different protocols and process replication updates differently. It is therefore not an option for a generic architecture to use a single propagation queue for replication messages. Instead of using a simple ordering in a single queue, we use multiple queues, a scheduling algorithm, and a replication architecture, where the scheduler chooses messages based on the properties of the replication methods for each message sent on the network (Section 4.4). There are several differences to a simple queue approach for replication: 1) Bounded time replication messages must always take priority, and must wait, at the most, one message delay before being sent at the network. 2) Updates for full consistency need not be timely, so such update propagations can be interleaved with best-effort update messages, where best-effort messages take precedence. Due to such interleaving, best-effort is given priority if queued. 3) Deferred update propagation messages are augmented with a time stamp. Time-stamped delays need a different queue management policy. The policy used for the deferred updates is application dependent, and our example
of scheduled replication uses the time stamp as the earliest time of propagation. When the current time passes the time stamp, the deferred update can be sent.
4.4 An architecture for segmentation

For the architecture of coexisting replication methods, we assume the following:

1. There is no inter-segment dependency order for updates to other nodes, so updates for different segments can be propagated in arbitrary order. This enables the replication mechanism to handle updates for different segments of different consistency classes concurrently. In case there are dependencies, transactions need to be handled in a single consistency class.

2. There is no global ordering of updates, but updates are ordered locally by priority of the replication method at each node. Several updates for the same segment (possibly arriving from different nodes) are not globally ordered. However, they are ordered for integration in a bounded and an unbounded queue. No other order can be established at the integrating side, instead updates are integrated as individual updates, and simultaneous integration of updates cannot occur.

3. There is a scheme for eventual consistency, such as PRiDe, to support independent updates of concurrently existing object replicas. Such a scheme finds concurrent and conflicting updates and eventually resolves them. All the information needed for successful integration is contained within an update message.

4. Updates are sent using point-to-point based broadcast (or multicast), which may be implemented as point-to-point communication or as hardware-supported multicast.

Bounded time replication constraints

Bounded time replication is the most restrictive replication class considered here, and to ensure time bounds an additional number of conditions must be met, as proposed by Matheis and Müßig (Matheis & Müßig 2003):

1. There is a minimum inter arrival time for transactions that require bounded replication. There is also a maximum burst size (the maximum number of arrivals within a time period) for arrivals of such
transactions. A lower bound on the inter-arrival interval enables us to limit the buffer size for updates to be propagated and thereby we get a predictable propagation delay time in this buffer.

2. There is a maximum for the number of nodes that can propagate bounded time replication updates to a node concurrently. For a bound in the integration time of such updates, there must be a bound on the incoming queue size. The queue size must be dimensioned on the basis of the assumed maximum amount of concurrent updates coming from other nodes.

3. There needs to be a bound on the integration time for bounded time replication updates taken from the integration queue, in order to be able to ensure a bound on the timeliness for integration of the remaining entries in the integration queue.

4. The bound on the delay for bounded time replication will depend on the available bandwidth and the network delay. The bandwidth needs to meet the bandwidth resulting from the sum of all the nodes using the same network segment, with transactions using the minimum inter arrival time.

5. Support for bounded replication requires a real-time network. Properties of such a real-time network are (Verissimo 1994): 1) Enforce bounded delay from request to transmission of frame; 2) Ensure that a message is delivered despite occurrence of omissions; 3) Control of partitioning.

The main criteria for the propagator is that any bounded time replication messages in the bounded queue must to be sent out on the network within a known time bound. Further, any other replication messages should not block each other. At most, our algorithm blocks a bounded replication update message the time it takes to send out a single message on the network. We use a semaphore for exclusive access to the network, and any message propagation, except bounded message propagation, returns the semaphore after each message sent. Thus, after any message sent to the network, control is
4.4 An architecture for segmentation

returned to $Q_{\text{bounded}}$, to give a maximum delay for the message at the head of $Q_{\text{bounded}}$ of the time it takes to deliver a single message to the network.

Replication updates can be divided into three main groups: 1) Updates with global timeliness bounds for the updates to be integrated into remote replicas. 2) Updates that need full consistency between replicas, but where there is no requirement of timeliness bounds on updates. 3a) Updates that can be delayed for applications that can tolerate eventual consistency, but without timeliness bounds on updates. 3b) Updates that are deliberately delayed to some future point in time, or some other criteria. This type utilizes eventual consistency to intentionally reduce communication to save resources. Delays and criteria for propagation must be designed since it is application dependent.

Note that we do not consider consistency models that require both full consistency guarantees or global timeliness guarantees for updates. Such systems are not in the scope of this thesis. They usually require fully specified applications and resource usage. Well-specified applications cannot easily tolerate execution outside of the specified operating envelope. It is difficult to achieve scalability for large-scale and fully consistent databases due to the synchronization required between nodes, so approaches for scalable systems need to use weak consistency schemes (Helal, Heddaya & Bhargava 1996).

Scheduled propagation

With scheduled replication for four consistency classes, propagation can be done by three threads. The EC propagation thread (Algorithm 4.1) manages propagation for all eventual consistency updates (Used by Immediate Consistency Replication, Best Effort Replication, and Bounded Time Replication). This thread has the highest priority and gets the semaphore before lower priority threads. It sends out the entire content of the bounded replication queue ($Q_{\text{bounded}}$) before returning the semaphore for other propagations. Other propagations are allowed to use the semaphore for single message propagations only, which is the case when propagating from other queues used for eventual consistency propagation, such as ($Q_{\text{effort}}$, $Q_{\text{deferred}}$). The 2PC propagation thread (Algorithm 4.2) uses a receiver
thread, to receive the confirmation replies from all other nodes in the two phases of message exchange required by the 2PC protocol.

```
while true do
    /*Two pending calls need to return: */
    queues.sleepUntilContent();
    network.sleepGetSema();
    /*Propagate all bounded time replication updates first */
    while qBounded.size() > 0 do
        network.sendOnePackage(qBounded.head);
    end
    if qEffort.size() > 0 then
        network.sendOnePackage(qEffort.head);
    end
    else
        if qDeferred.size() > 0 then
            if now > qEffort.head.time() then
                network.sendOnePackage(qDeferred.head);
            end
        end
    end
    /*Extend list for other single-message EC updates */
end
```

Algorithm 4.1: Scheduled replication: EC Propagation Thread

Scheduled integration

The Integrator component receives updates from the network and forwards them to the EC integrator or the 2PC integrator, depending on the type. For EC integration, the architecture needs to separate bounded and best effort integration messages. The bounded integration queue $Q_{bounded}$ is bounded in size, and the size need to be large enough to hold the number of expected incoming concurrent bounded propagation messages from a bounded number of nodes (following the constraints in Section 4.4.1). Note that there is bound on the number of replicas (with a typically low number of replicas) in a ViFuR database, why the queue needs to store concurrent updates only from a bounded number of nodes. The queue must be large enough to
4.4 An architecture for segmentation

store all such updates, since there is no way of rejecting bounded replication messages at the integrating node. Once bounded update transactions are admitted at the propagating node, there should be no way of delaying them in the replication architecture. Instead, replication must guarantee timely delivery, and updates must be received at the integrating node at the rate of the arrival of update messages from the network.

A single queue can be used for both best effort and deferred integration messages, $Q_{ff}$, since integration is not time critical; there is no time bound to meet for the delay from the admission to the finalized integration. The

```c
/* 2pcPropagator() thread */
nodelist = segmentManager.getNodelist(q2pc.head());
numnodes = nodelist.size();

while true do
    q2pc.sleepUntilContent();
    network.sleepGetSema();
    /* Phase 1, Agreement for update */
    if r2pcReceiver.coordinated() == false then
        network.sendCoordRequest(nodelist.get(sent));
    end
    /* Phase 2, Engage update */
    if r2pcReceiver.coordinated() == true then
        network.sendOnePackage(q2pc.head());
    end
    network.releaseSema();
end
/* r2pcReceiver() thread */

while true do
    network.sleepUntil2PCReply();
    if replyCount == numNode and coordinated = false then
        coordinated := true;
    end
    else
        if replyCount == numNode and coordinated = true then
            coordinated := true;
        end
    end
end
```

Algorithm 4.2: Scheduled replication: 2PC Propagation Thread
size of the queue must still be large enough to store the number of expected concurrent updates to be received from other nodes, since also such updates are both admitted and committed.

Updates of $Q_{bounded}$ and $Q_{fifo}$ queues are integrated individually into the database. The integration of eventually consistent updates are order-independent, and updates become valid ones in the database, when committed at the propagating node. The consistency management scheme ensures that updates eventually become available at all replicas. For $Q_{bounded}$, updates may be delayed at most the time needed to process the entire content of $Q_{bounded}$. For this reason the size of the queue must be bounded. For $Q_{fifo}$, delays of updates will delay the stabilization time for consistency in the same way as any delay that occurs between a commit and the processing by the Resolver. With 2PC integration, the first phase locks objects to be updated. The integrator then responds positively to the propagating node if objects to be updated can be locked. At the second phase, the database is updated and the update is committed.

4.5 Conclusions

This chapter introduces segmentation as a scalable approach for grouping and management of objects based on their properties. We provide formal definitions for segmentation, and show how data properties are used for multiple segmentations of a database. Further, we introduce the usage of rules to achieve consistent segmentation for multiple data properties. Also, rules enable consistency with application requirements that are not expressed by the data properties, and for also enable consistency with the execution infrastructure and environment. Finally, we provide a model database system architecture that supports segmentation, and which allows multiple consistency classes to co-exist in the same replication architecture.

4.5.1 Results, contributions and subproblems met

In previous work, Virtual Full Replication (ViFuR) was suggested but not pursued (Andler et al. 1996). In this chapter, a formal definition of seg-
4.5 Conclusions

mentation is provided for ViFuR that captures how objects are grouped into segments, and which constitutes the basis for the subsequent work on segment management. We introduce a segment-based management structure that holds segment properties, such as object identifiers, nodes, and other properties. In addition, we show how rules for property consistency in segmentation are applied in such a structure. By elaborating on database segmentation and providing a formal framework, we solve subproblem 1a and 1b.

The model architecture for segment based replication allows multiple consistency models to co-exist. We present features that are needed in such an architecture, as well as requirements to be fulfilled for it. The architecture is the basis for the work in subsequent chapters, and solves subproblem 1d.

4.5.2 Discussion

The segmentation of a database for ViFuR is one management approach for creating a perception of full replication. However, alternative approaches also need to consider the scalable management of data objects that have multiple properties. Such scalable management of data objects needs to consider multiple ways of using data, for example where to allocate it, and how to manage consistency between data replicas in the architecture used.

In our work, we have used four common replication methods: for immediate consistency, best effort, bounded time, and for deferred replication. These four replication methods meet the needs of most consistency models that may be of interest for distributed real-time databases. The model architecture is designed for extensibility, to meet the requirements of additional replication methods. The architecture may be extended with additional queues, and with different usage of the queues for replication and integration. The actual usage with additional replication methods depends on the consistency model required for an application and the model architecture can be extended accordingly.
Chapter 5

Static Segmentation for Fixed Object Allocations

_Status Quo, you know, that is Latin for "the mess we’re in."
- Ronald Reagan_

In this chapter, we pursue Virtual full replication with static segmentation used in a distributed real-time database (Mathiason et al. 2005). We propose an implementation-approach of static segmentation that segments a database based on multiple data object properties, known prior to the execution of the system. Object properties and dependencies between such properties are used to form segments. Static segmentation, as presented in this chapter, is based on fixed property settings that do not change during execution. In the following chapters, we present how adaptive changes can be made to such static segmentation of the database during execution. Static segmentation optimizes the allocation of replicas, by using the read and write sets of transactions as a specification of the data needs. In a typical system only a few replicas are needed, and a bound on the number of replicas can be derived from the specification of data needs. With a bound on the number of required replicas, concurrent resource usage is bounded. We use the database segmentation approach, as presented in this chapter, to segment a database on a set of selected properties and achieve bounded resource usage for scalability. Further, multiple segmentations can be generated for multiple purposes usages. Objects with identical or compatible properties
can be combined into segments to reduce the number of segments.

5.1 Implementing static segmentation

Based on the principles of segmentation as described in Chapter 4, this chapter presents an approach for static segmentation that uses multiple properties and matches them for grouping objects into segments. For the implementation of this approach, a matrix data structure is used that organizes properties and their settings prior to execution, where an algorithm can efficiently compare and adjust property settings, and to apply rules. Once rules are applied and dependencies are consistent, objects can be grouped efficiently for segments. In order to simplify the presentation, we start by using a sparse two-dimensional matrix, to list the object properties and their settings. Later, we address the memory usage of such a sparse matrix, and provide a dense representation of the same information, which uses a fraction of the memory needed for a sparse matrix.

For a matrix of properties, database objects are listed as rows of the matrix, and properties, such as the required nodes of allocation, are listed in columns. For a segmentation on allocation, known data accesses are marked out. There is no need to consider differences between read and write accesses. In the case of a quorum-based approach, the type of access is essential information to manage quorums, while for ViFuR we must have a local replica, regardless of the type of access. The marks in the matrix are singleton sets, therefore, additional access marks for access of the same objects at the same node are not counted.

Consider again the example in Section 4.2.3. Figure 5.1 (left) shows the matrix with 6 objects replicated to 5 nodes. Here, we segment on allocation of data objects only, and we expand below the example for segmentation on multiple properties. The property of allocation is essential for ViFuR, since its presence is a condition for other properties to be valid at the node. Since other properties require an allocation, we regard allocation to be the most important object property used for segmentation. For this reason, we focus on allocation for scalability in chapters 6 and 7, while other properties can still be used.
5.1 Implementing static segmentation

In the matrix, each column of properties and their settings is represented by a binary value, so that each row can be interpreted as a binary number. The binary number for each row is used as a key to uniquely identify the nodes where the object is needed, based on known transactions. Thus, the pair <object id, key> represents the object and its properties. The objects needed for access at the same set of nodes will get the same binary key. We use the key to group objects together, and by first sorting the matrix rows on the key, we get the table in Figure 5.1 (right). Then, by stepping through the matrix only once and comparing the keys, we collect the rows that have the same key into the same segment. Key comparisons only use the key at the current and the next rows for this. Algorithm 5.1 gives the pseudo code for the generation of segments.

Algorithm 5.1 is our basic algorithm for static segmentation prior to execution (off-line segmentation). The sort operation in this algorithm contributes the highest computational complexity to it, such that the algorithm segments the database in $O(o \log o)$, where $o$ is the number of objects in the database. It gives an efficient algorithm for an off-line segmentation for known accesses that can be pre-specified, but does not easily allow changes of the set of known accesses. If this set changes, the entire algorithm needs to be re-executed.

Figure 5.1: Accesses and keys (left), and resulting segments (right)
/*Fill out property settings, fill out reads and writes of transactions  */
clear(access);
for i ← 1 to numtransactions do
  for j ← 1 to T[i].numnodes do
    for k ← 1 to T[i].L.numreads do
      mtx(j,T[i].L.read[k])=1;
    end
    for k ← 1 to T[i].W.numwrites do
      mtx(j,T[i].W.write[k])=1;
    end
  end
/*Setup binary keys for properties  */
for j ← 1 to numnodes do
  mtx.colKey[j]=2^j;
end
/*Calculate key  */
for i ← 1 to numobjects do
  mtx.Key[i] = BuildKey(access,i,numnodes);
end
/*Sort rows on key  */
SortMatrix(mtx.Key, mtx);
/*Find segments and allocations  */
segmID=0;
currKey = -1;
currSeg = NIL;
for i ← 1 to numobjects do
  if currKey /= mtx.Key(i) then
    currKey = mtx.Key(i);
    currSeg = NewSeg(Key);
  end
  currSeg.Add(mtx.oid(i));
end

Algorithm 5.1: Single-property segmentation (property of allocation)
5.1 Implementing static segmentation

5.1.1 Static segmentation on multiple properties

By adding columns to the matrix, it can be extended for the segmentation on multiple properties. To use multiple properties we extend the first part of Algorithm 5.1, where properties are marked out. The extension of the matrix adds as many columns as the number of possible values that each property added requires. For properties that are mutual, the representation uses the same number of columns as the number of values, while a non-mutual property may use multiple columns for a cardinality that may be many times more than the number of columns.

Assume, as an example, that we add the data properties of consistency model and storage medium to be used as objects in the database, and that we add these to the matrix of accesses in the example of Figure 5.1. This gives us an extended matrix with addition of new columns, and where each new column is assigned a unique binary value (Figure 5.2). With this extended matrix, unique segments based on all the properties can be generated, still with a computational complexity of \( O(n \log n) \).

![Figure 5.2: Multiple property matrix](image)

An unrestricted combination of properties may potentially result in a large number of segments (Equation 4.1). For the example, the possible number of combinations of nodes, the cardinality, is \( 2^5 - 1 = 31 \) segments for 5 nodes. This is the sum of \( r \)-combinations for node combinations 1..5, or \( \sum_{k=1}^{5} C(n, k) \) where \( n \) is the total number of nodes and \( k \) is the seg-
\[ C(n,k) = \frac{n!}{(r!)(n-r)!} \]
and thus the cardinality of allocation is 31. The property of consistency is a mutual property, so that only one setting can be used at a time. In this example, there are three consistency types, \( c = 3 \), so the cardinality of consistency is 3. The property of medium is a mutual property as well. In this example, \( c = 2 \) for storage medium, so the cardinality of storage medium is 2. For these properties combined, there are \( 31 \times 3 \times 2 = 186 \) possible segments. For the 6 objects used in the example, there will be no more than 6 segments, but in a database with more objects, these properties may create 186 segments.

### 5.1.2 Applying rules for property consistency

With multiple properties and their required settings listed in the matrix, we can now apply rules, as introduced in Section 4.3.3, for validating the consistency between properties. To reduce the number of segments, we consider the properties and their relations. Our example uses three properties: allocation, consistency model, and storage medium. There are three types of dependencies to consider (Section 4.3.3): "Dependencies on infrastructure properties", "Dependencies between data properties and their settings", and "Explicit data properties". Now, assume that the architecture provides a real-time network, and that the requests will not exceed the bandwidth of the real-time network, so that we can ensure bounded time replication. For this example, we have solved the dependencies on the infrastructure. Also, assume that we chose to adjust inconsistent dependencies between properties, such that we allocate \( o_5 \) in main-memory (instead of disk, as originally required) to allow bounded replication. That solves dependencies on other properties for this example.

In addition to the consistency relational rules introduced in Section 4.3.3 there are additional rules originating from the application requirements: 1. For fault tolerance by redundancy it may be required to add ghost replicas to ensure a minimum degree of replication of data objects, 2. Explicit clustering of groups of objects by setting the properties equally, and 3. Pinning of objects to nodes to disallow deallocation of replicas. Instead of using these type of rules in the example presented above, we describe how such rules
5.1 Implementing static segmentation

can be applied to the matrix of objects and properties:

1. Additional replicas are introduced, and this can automatically be done by changes of the property matrix during rule evaluation. Additional marks are filled into the matrix at nodes with no current replica of the object, following some replica allocation policy. The allocation policy for adding ghost replicas is application dependent or dependent on the actual execution environment. Typically ghost replicas are assigned to nodes with high availability or bandwidth.

2. Lists of clustered objects are set to have the same property settings. Such a change to settings must include a consideration of the consistency rules. The alignment of settings will assign objects to the same segments during segmentation.

3. Pinning is an approach that prevents objects from being removed from nodes at execution time. Our adaptive segmentation approach in Chapters 6 and 7 enables the usage of pinning. Allocations used by hard real-time transactions need guaranteed availability, and thus must be pinned. Pinning can be implemented as an extra column in the matrix, and such pinning will become a part of the segment information.

5.1.3 Multiple segmentations

The matrix of collected object properties and settings may not only be used for single-segmentations. With consistent properties in the matrix, we can create multiple segmentations on the set of objects and the properties with the settings as shown in Table 5.2. A segmentation is done on a selected set of properties, $P$, and has a certain purpose, as we see in the following segmentation examples.

- A segmentation on the properties which will give physical units of allocation of the data objects use $P = \{P_1, P_2\}$, where $P_1 = \text{allocation}$, and $P_2 = \text{storage medium}$. We call segments in such a segmentation, which divide the database into physical allocation units, physical segments. The specific segmentation for physical segments in the example is identical to the segmentation of the single-property example in
Section 4.2.3. The difference is that the single-property example segmentation originates from a set of all the properties, where rules have been used for a validation of other properties, the execution environment, and with specific application rules.

- A different segmentation on the same matrix, to be used by the replication module for grouping updates for propagation, needs a segmentation on \( \mathcal{P} = \{P_1, P_2\} \), where \( P_1 = \text{allocation} \), and \( P_2 = \text{consistency model} \). With this kind of segmentation, the objects that can be sent out together in a block update can be replicated together. Such segments are not stored at some physical location, but are used as a logical entity by the Replication manager. We call these segments logical segments.

- A third segmentation in the example matrix could be the groups of objects that should be recovered after a node failure, and where objects are loaded together to the node in a data block, to reduce the copy overhead. The recovery of some segments before others allows certain transactions to start before other segments are (entirely) recovered if needed, since a transaction that has all its objects locally can start to execute. For such a segmentation only allocations of objects are needed, so \( P = \{P_1\} \), where \( P_1 = \text{allocation} \).

### 5.2 Efficient coding of segments using Bloom Filters

The coding of properties using a sparse matrix is a simple way of keying property settings for the grouping of objects, and was used in our first version of the implementation of static segmentation in the DeeDS prototype. Using many nodes, this sparse matrix implementation will soon lack a wide-enough representation. This section presents a remedy that provides a compact representation of properties.

A sparse matrix needs to use much storage compared to the number of entries that actually carry information of property settings. In particular for non-mutual properties, the bit representation is soon depleted of bits in the representation of any variables that can hold the bit positions in the binary number required for a large number of nodes. Both for storage space and
5.2 Efficient coding of segments using Bloom Filters

for the ability to represent segments, we need a more efficient storage for creating keys.

One possibility for efficient storage of non-mutual property settings is to use hashing functions for a key to the unique combination property settings. However, hashing of data sets may give the same hash key for different data sets that are hashed, such that there is a risk for hash conflicts. A hash conflict results when two different sets of data are hashed to the same hash key. The hash key will not point to a single set of data, but it will be an ambiguous hash key. Hashing schemes that use multiple hash functions reduce the risk, and one of such schemes is the Bloom Filter (Bloom 1970). Bloom Filters use multiple and independent hash functions for set membership decisions, in particularly in distributed applications, where nodes can decide set membership, without an explicit agreement between nodes. Bloom filters never give false negatives, but they may give false positives for set membership queries.

Using Bloom Filters will reduce the need of large bit-string representations. Instead, only the useful information is stored in the property table. For instance, a bit string that represents storage at nodes 1 and 280 would use a 280-bits wide binary value, while the representation with Bloom Filters instead uses an set of two integers. The larger the system, the more the benefit with this dense representation based on Bloom Filters.

Bloom Filters use \( k \) independent hash functions \( h_1, h_2, \ldots \), to hash an element (or a data set) \( a \in A \), where \( A \) is the set of elements that belongs to the set. The set of hash functions maps \( a \) to a bit vector \( v \) with \( m \) bits. For an element \( a \in A \), the bits in \( v \) at positions \( h_1(a), h_2(a), \ldots, h_k(a) \) are set to 1. With a query for an element \( b \) for membership in \( A \), we read the bits at \( h_1(b), h_2(b), \ldots, h_k(b) \). If any of these bits are 0, \( b \) is not in the set, otherwise, we conjecture that \( b \) is in the set. Due to possible false positives, there is a chance that \( b \) is still not in the set. The probability of false positives is approximately \( (1 - e^{kn/m})^k \), after that \( n \) keys have been inserted. Both the insert and membership operations execute in \( O(k) \).

Using Bloom Filters to code object property settings for segmentation, objects with the same property settings will have the same key, \( v \), which codes a set of property settings. Bloom Filters generate the same bit vector
for the same combination of property settings (when using the same set of hashing functions, which is assumed). Thus, the key can be used to group objects into segments in the same way as with a bit representation (Section 5.1).

Compared to a single hash function, the use of Bloom filters reduces the risk of hash conflicts significantly. Nevertheless, conflicts may still occur. Our segmentation algorithm is updated to handle such conflicts. The property table, used for off-line segmentation on known property settings, stores both the key and the properties. The key allows the sorting of objects on the property settings, while the settings in the property table are used to store the properties once only for each segment. Using Bloom filters for static off-line segmentation requires a change to Algorithm 5.1. The algorithm handles hash conflicts and distinguishes objects with different property settings that have the same key, and separates such objects into separate segments. The updated algorithm compares the actual setting for each key and creates multiple segments from the objects when different settings are found for the same key (Algorithm 5.2). Since it must be assumed that objects with the same key may be unordered, a temporary list of segments that are in creation need to be used. The list is used as long as the same key is found when stepping through the list of objects. Once a different key is found, all the segments in creation can be saved and all the temporary list can be removed. The old key will not appear again, since the table is sorted according to keys. With our algorithm, the fast membership operation of Bloom filters is not utilized. Instead, the main advantage obtained from Bloom filters is the low hash conflict rate as a consequence of using multiple independent hashing functions.

5.3 Distribution of static segments

The segments that are setup by the static segmentation algorithm are distributed to the nodes of the system before execution can start. Note that the entire segment list is not needed at all the nodes of the system. For the assumed type of application, a few large-granular segments are needed at each node. Segments that list the physical allocation of objects (physical
5.3 Distribution of static segments

```
/*Fill out property settings and create keys*/
...
/*Sort property table rows on key*/
SortMatrix(mtx.Key, mtx);
/*Find segments and allocations*/
segmID=0;
currKey = -1;
currSeg = NIL;
usedSettings = NIL;
for i ← 1 to numObjects do
  if currKey != mtx.Key(i) then
    currKey = mtx.Key(i);
    usedSettings = NIL;
    currSeg = NewSeg(createUnique(Key), mtx.settings(i));
  else
    if contains(mtx.settings, usedSettings) == false then
      currSeg = NewSeg(createUnique(Key), mtx.settings(i));
      add(mtx.settings, usedSettings);
    else
      currSeg = getSegment(mtx.oid(i));
    end
  end
  currSeg.Add(mtx.oid(i));
end
```

Algorithm 5.2: Multiple-property segmentation using Bloom filter hash keys

segments) are sent to the nodes where the allocation is needed, and logical segments are sent to nodes where there are physical segments.

Transaction execution at a node can start as soon as the objects needed have been instantiated at the node, after the segments are available at the node. However, not all replicas at a node need to be available at other nodes to run the local transaction. Each local data replica can be used without synchronization with other replicas (because of the eventually consistent replication scheme). Once other replicas become known, update propagation starts, and the replicas will have eventual consistency.
Incremental recovery is a suitable mechanism with which to establishing replicas (and also entire physical segments) from existing replicas at other nodes. It can be used as soon as the locations of other replicas have become known. To enable incremental recovery from another node, there needs to be another replica assigned to a physical segment, either by an off-line segmentation or by an on-line creation of a first replica at some other node. The setup of data objects by incremental recovery is further pursued in later chapters, where the allocation of segments is allowed to be adaptive, and replicas can be set up and also removed dynamically.

5.4 Analysis

In order to understand the cost of segmentation, we analyze the segmentation overhead, as well as the resource usage for a segmented database in terms of bandwidth, storage and the processing time spent on nodes to manage segmentation.

5.4.1 The cost and benefits of segmentation

The segmentation of the database introduces management storage structures containing meta-data about the objects and their properties. This is stored at each node, for the objects with replicas at the node. The storage used does not directly contribute to the storage of database values, but is an overhead spent to manage resource usage in the distributed database.

The overhead for segmentation of a database needs to be known. The overhead cost per object needs to be small, compared to the benefits in total storage that segmentation gives. By quantifying the size of the storage overhead, we will know the usage parameters where segmentation is worth the overhead. To simplify the analysis for presentation, we first use data allocation to nodes as the single property for segmentation. The analysis is later expanded for multiple properties.
5.4 Analysis

5.4.2 Resource usage in a segmented database

We evaluate the approach by examining how three important system resources are used when selected system parameters are scaled up. The baseline for comparison is a fully replicated database with detached replication, such as DeeDS. A large-scale distributed database with many nodes requires scalable bandwidth usage. Full replication requires $O(n^2)$ replication messages for $n$ nodes. Also, a database with full replication stores the entire database on the local node, requiring storage of $O(n)$ data objects. Conflict detection and resolution need $O(n^2)$ processing time, since every node that has a replica must resolve a conflict caused by any other node. Based on our database model, we present an analysis in the following section about how these three key system resources scale for ViFuR. Since full replication is can be regarded as a special case of ViFuR, but which has a replica at every node, the analysis model can be used for resource analysis of full replication as well.

Analysis model and assumptions

We assume that the database is distributed to $n$ nodes, and with segmentation the degree of replication is limited to $k \leq n$ nodes. In such a database, every update is replicated to $k - 1$ nodes. With network functions for multicast or broadcast, the system could replicate data specifically to a set of nodes by a single send message, without reducing the integration time or the conflict detection and resolution time of an update. In this section we choose to analyze ViFuR using a network that provides point-to-point communication. We argue that an analysis based on point-to-point communication is a general case that can be applied in most networks, and that it also represents a worst case for the resource usage. Networks with multicast services enable optimizations to resource usage that may give large improvements. However, the case worst resource usage is important for a scalability analysis. Selective replication by multicast to groups of nodes (Alonso 1997) is a scalability approach that is orthogonal to ViFuR. We intend to use multicast propagation in future work as part of a study that involves more complex network topologies than the network used in the current analysis model. We
also consider coordinated propagation of replication messages, including push-pull techniques, to be a future extension of our work.

We consider each object update to generate a send of \(k - 1\) single update messages. Our simple analysis model in this section does not consider individual degrees of replication \(k_i\) for each segment, but the same degree of replication for all segments, \(k\). The number of transactions in the database is denoted \(t\). We characterize each transaction, \(T_j\), by its frequency, \(f_j\), the size of its read set, \(r_j\), the size of its write set \(w_j\) and the size of its conflict set, \(c_j\). Thus, the characteristics of each transaction can be described as \(T_j = \{f_j, r_j, w_j, c_j\}\).

A scalability analysis of network usage is an essential component in a general scalability estimation for a system. The network is an important shared system resource in a distributed system. We choose to model bandwidth as a single shared network segment, as is the case with traditional Ethernet or simple time-slotted network communication. A single shared network resource is more restrictive for scalability than a switched network, for instance, where the design topology can relieve congestion by distributing communication on multiple independent network links.

A segmented database stores \(k\) replicas of database objects, instead of \(n\) replicas that are stored in a fully replicated database. For applications that typically have a low replication degree, \(k\), segmentation can save much memory overall and improve scalability. To analyze storage we consider the storage required for database object replicas.

The processing time for storing updates on the local node includes time for locking, updating and unlocking the data objects, which we denote \(L\). Updates for data objects that have replicas on other nodes use additional processing time to propagate the update, including logging, packing, marshalling, and sending the update on the network. We denote the processing time for propagation \(P\). With point-to-point communication, the sending node spends \(P\) time for each node that has a replica of the updated data object. At the node receiving a replicated update at another node, the database system must integrate the update. This includes the receiving, unpacking, unmarshalling, and locking of data objects, as well as detecting and resolving conflicts, updating of database objects, and finally unlocking the
5.4 Analysis

Data objects that were replicated. We denote the time used for integration $I$, excluding conflict detection and resolution, while the time used for conflict detection and resolution is denoted $C$. Processing time $C$ only occurs when there are conflicting updates, while $I$ is used for all updates replicated to a node. Integration processing is needed at all nodes having a replica of the originally updated data object.

Example of worst case segmentation storage overhead

The overhead for static segmentation of a database depends to a great extent on how data is accessed. Segmentation overhead depends on the number of replicas using the same property settings. Data that is highly cohesive will likely be accessed together at several nodes, resulting in relatively large segments, as large-granular segmentation. Unrelated accesses to data over several nodes will generate smaller-granular segmentation since data is not related.

To determine the overhead of segmentation for allocation we compare with no replication, which uses no segmentation management overhead. Consider a vehicle application using a distributed real-time database for exchange of variables between processors (ECU’s) (Nyström et al. 2002). Database objects are equally and disjointly distributed over the nodes, and typical parameters for an evaluation of realistic usage are listed in Table 5.1.

<table>
<thead>
<tr>
<th>Standard Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database nodes</td>
<td>50</td>
</tr>
<tr>
<td>Database size</td>
<td>50 objects/node</td>
</tr>
<tr>
<td>Number of replicas</td>
<td>1–50</td>
</tr>
<tr>
<td>Data object size</td>
<td>Uniform (1–128) bytes</td>
</tr>
<tr>
<td>Object ID storage</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Object ID to segment reference</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Storage of property value</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Table 5.1: Parameters of the example

As an example of the cost of segmentation overhead, consider the example: 50 nodes are populated with 50 objects per node with an average size of 64 bytes/object. An object ID uses 4 bytes, and a segment reference uses 4 bytes. For a single-property segmentation using 4 bytes to store the
property, we increase the degree of replication from 1 to 50. Each segment will start at the same node as with the first replica, such as \{n_1 : o_1, ..., o_{50}\}, \{n_2 : o_{51}, ..., o_{100}\},... \{n_{50} : o_{2451}, ..., o_{2500}\} and the degree of replication is 1, since there is only one replica of each object in the database. Increasing the degree of replication to 2 gives \{n_1, n_2 : o_1, ..., o_{50}\}, \{n_2, n_3 : o_{51}, ..., o_{100}\},... \{n_{50}, n_1 : o_{2451}, ..., o_{2500}\}. At a replication degree of 49, there are 49 segments containing the node list of all the nodes except one. Finally, increasing the replication degree to 50 (full replication of all objects), there will be one segment that lists all the nodes, and the segment list is \{n_1, ..., n_{50} : o_1, ..., o_{2500}\}.

The overhead of these segments is a worst case segmentation, since all the segments are disjoint. It is not a typical usage scenario, but indicates the worst case overhead. Increasing the (required) degree of replication in this way will generate a growing number of segments with an rising overhead that cannot be joined until the very last configuration of 50 replicas. The overhead for this worst case example will increase from 13 to 19% compared to the amount of total data stored. We further evaluate the overhead of segmentation focusing on adaptive segmentation, in Section 6.3.

Analysis of resource usage

We now analyze the resource usage of three critical system resources. Scalability is estimated by examining how usages of these resources scale with the degree of replication of objects.

Bandwidth usage depends on the number of updates replicated, including the new values and their associated version vectors. In our model there are \(t\) transactions, and every transaction \(T_i\) execute at frequency \(f_i\), generating one network message for each member of its write set \(w_i\) to update its replicas at \(k - 1\) nodes. Such network messages are generated by all the transactions at all the nodes. We express bandwidth usage in formula 5.1.

\[
(k - 1) \sum_{i=1}^{t} w_i * f_i \quad \text{[messages/sec]} \quad (5.1)
\]

The formula indicates that bandwidth usage scales with a factor of the degree of replication, \(k\). For a virtually fully replicated database with a
5.4 Analysis

bound $k$ for the degree of replication, bandwidth usage scales with the number of replicas, $O(k)$, rather than with the number of nodes, $O(n)$, as is the case with a fully replicated database. However, the number of transactions often depends on the number of nodes, $O(n)$, bandwidth usage therefore becomes $O(kn)$ for the virtually replicated database and $O(n^2)$ for the fully replicated database.

For the storage of a database with $s$ segments, there are $k_i$ replicas of each segment of the size $s_i$. We express required storage for a virtually fully replicated database in formula 5.2.

$$\sum_{i=1}^{s} (s_i \times k_i) \text{ [objects]} \quad (5.2)$$

In our analysis model, we assume the same degree of replication, $k$, for all segments, $\forall i (k_i = k)$. Thus, each data object $o_i$ in the database of $o$ objects is replicated at $k$ nodes and the required storage can be expressed as $o \times k$ for the number of objects stored. With full replication to $n$ nodes, $n$ replicas of each data object is stored and the required storage is $o \times n$ objects.

A virtually fully replicated database scales with the bound on replication, $k$, rather than with the number of nodes, $n$, which is the case with a fully replicated database.

The processing time used for replication depends on the write set of each transaction, $w_i$, resulting in propagation time, $P$ and integration time, $I$, for each node to replicate to. Additionally the conflict set, $c_i$, requires conflict detection and resolution time, $C$. Thus, we can express the processing time as $\sum_{i=1}^{t} f_i \{[L + (k - 1)P + (k - 1)I]w_i + [(k - 1)C]c_i\}$, or as formula 5.3.

$$\sum_{i=1}^{t} f_i \{Lw_i + (k - 1)\{(P + I)w_i + Cc_i\}\} \quad \text{[sec]} \quad (5.3)$$

Similar to formulae 5.1 and 5.2, this formula shows that processing is constant with the degree of replication, not growing with the number of nodes in the system. Also, the sizes of the write set and the conflict set influence the amount of processing time required.
5.5 Conclusions

This chapter presents how to implement segment management structures for pre-specified data needs, and describes a scalable off-line segmentation algorithm that supports segmentation for single or multiple properties.

5.5.1 Results, contributions and subproblems met

The implementation of the table-based management of data properties allows the segmentation of a database on multiple properties in $O(o \log o)$ time at a storage of $O(o + s)$ when the table is coded using Bloom-filters, and where $o$ is the number of objects and $s$ is the number of segments. The number of segments is application-dependent, such that for applications with many cohesive objects that share the same property settings, there will be fewer segments than for applications with less cohesive objects. With a table-based segmentation, rules can be efficiently applied, so that segmentations generated are consistent with data properties, the application and the execution environment. Our efficient table-based approach solves subproblem 1c.

The analysis of the usage of three key resources for scalability, storage, bandwidth and processing time, reveals that the cost of resource usage when applying segmentation increases linearly with the number of replicas required in the system to meet the pre-specified data needs. Resource usage does not increase with the number of nodes or the size of the database, only with the concurrent need for availability of data. Our analysis completes the solution for subproblem 1d (also met by the model architecture presented in Chapter 4).

5.5.2 Discussion

The table-based approach using bit keys to code property settings is very costly in terms of storage, and it is used for an initial description of the principles of the table-based approach. With a Bloom-filter, the table-based approach behaves as a set-based management of segments. The following chapters show how to manage segments as sets for incrementally changing
5.5 Conclusions

segments directly, rather than building up a table of properties to segment the database based on such a table.

The bandwidth analysis in this chapter assumes point-to-point communication between database nodes. Such replication is more costly than group-based (or broadcast-based) replication, where the bandwidth usage could be much lower for databases with a high degree of replicas. For our evaluation, we choose to model bandwidth using point-to-point replication, since, for the type of applications we intend, with a bounded set of replicas, the benefits of using group-based replication is not as large as with systems using full replication. Also, not all communication protocols that could be used support group based communication. Using group-based replication may save bandwidth for fully replicated databases, but will not reduce the amount of storage required at each node at all.
Static Segmentation for Fixed Object Allocations
Chapter 6

Adaptive Segmentation for Dynamic Object Allocations

There is nothing so stable as change.
- Bob Dylan

In this chapter we present an approach for ViFuR that uses adaptive segmentation (ViFuR-A), and incrementally changes the segmentation of a distributed real-time database, to support new data needs by re-allocations of replicas during execution time. ViFuR by static segmentation (ViFuR-S, Chapter 5) uses pre-specified data needs for static allocation of data object replicas. Static segmentation allocates objects based on the union of all accesses expected for a node over time, regardless of periods of access patterns, and such allocation typically causes excessive number of replicas, for an application with changing access patterns. Using adaptiveness, the set of local replicas can be changed to match the current needs at every moment, which generates a close to optimal allocation of replicas over time. With such adaptation, there is no need to allocate the union of all the accesses expected. Instead, only the objects currently needed are made available at the local node.
6.1 The need for adaptive segmentation

A fully replicated database with detached replication is independent of network delays, and allows timely local execution of transactions. A fully replicated database also comes with a very high cost in resource usage, due to irrelevant replication of updates. Pre-specification of the actual data needs enables scalability by bounding resource usage, under the assumption that the actual data needs and the required degree of replication, are bounded. Static segmentation is done prior to execution and makes data available only for data needs that can be pre-specified. Also, using static segmentation for scalability, the database loses the flexibility of full replication that allows the execution of arbitrary transactions at arbitrary nodes. Instead, transactions can only access data according to the pre-specification.

By introducing detection and adaptation to the current data needs, flexibility can be regained, while scalability can be maintained over time. Typical adaptations occur when the mode of operation changes for the clients using the data of the database, or when external stimuli patterns change. After a mode change, other transactions may execute instead, or transactions may execute differently, such that other data objects are read and written. New replicas may be needed for timely execution of transactions with data at the local node, while other objects at the node are not being accessed anymore and will consume resources without being used. We consider three levels of adaptations that need adaptive changes to the replication schema:

1. **Extensions of static segments.** An object is added to an existing static segment. Adding a new object to an existing local segment will cause no unpredictable delay to the transaction being executed, and the new object can immediately be used at the node after it has been initiated. Such objects are given a new global object identifier based on the creation node combined with a strictly increasing local counter at the node. No objects are allowed to be removed from segments.

2. **Migration of static segments.** A missing object is detected at a node of execution, and the entire segment to which it belongs is set up to the node, since objects in the same segment are closely related. To load a
6.1 The need for adaptive segmentation

segment, the node needs to find another node that has a replica of the segment needed, and a replica of the entire segment is made available. Segments are static, but the allocation of a segment may change.

3. **New allocations for individual object replicas.** A missing object is set up individually to the node where needed, rather than as an entire segment. Re-segmentation may occur with new allocations (in contrast to adaptations 1 and 2) based on the current needs for individual objects. Both the addition and the removal of objects are allowed, and re-segmentations control resource usage by optimal allocation of replicas for the currently known data needs. To set up an object, the node needs to find another node that has a replica of the object required. This type of adaptation allows several changes to occur at the same time for a set of data objects as well, effectively implementing adaptation type 2.

In order to find the current allocation of a missing replica in adaptation types 2 and 3, there is a need to handle global and dynamic replica allocation information, and the management of such information needs to be scalable. In this section we focus on adaptations of the replication schema by adaptive changes to segmentation, to meet unspecified requirements for the creation of new objects, allocations of new replicas of existing objects, and the removal of replicas. This covers adaptation types 1, 2 and 3. With these adaptations, we focus on the property of allocation of replicas, because this single property is a prerequisite of using an object of the database locally, and also for making use of other properties. Utilization of the key resources of bandwidth, storage, and processing time is directly influenced by the number of replicas used. Efficient allocation of replicas to meet the actual need of data is essential for maintaining scalable resource usage over time.

We present an adaptive approach for segmentation, and an evaluation of the scheme, using a detailed simulation study. Furthermore, we consider a system with up to several hundred nodes, in which database clients publish information in the database and ViFuR makes published information logically available at all the nodes. All accesses to the database are made by means of transactions. This kind of data sharing, by publishing
objects in a distributed database, resembles a distributed whiteboard. Global object identifiers allow distributed object sharing, and the first appearance of an identifier creates a new database object in the database. We assume that the network guarantees delivery and can assign unique node identifiers when new nodes are added. We take: independent updates, replication, conflict detection, and conflict resolution approaches of DeeDS, as described in Section 2.2, and ViFuR by static and off-line segmentation, as presented in Chapters 4 and 5.

6.2 ViFuR-A

The approach of Virtual Full Replication by Adaptive segmentation (ViFuR-A) includes distributed segment change management algorithms and a communication protocol, for the allocation (establishment) and de-allocation (removal) of replicas. The intention with ViFuR-A is to maintain scalability over time by resource management such that the number of physical replicas does not exceed the actual need, while ensuring transaction timeliness by the local availability of replicas. ViFuR-A makes the database scalable under the assumptions that the number of concurrently actively used data replicas is bounded, and where the rate of changes to the allocation of replicas is bounded as well. In ViFuR-A, replicas are made locally available for database clients, by considering the actual need. This approach shares some properties with other caching approaches for databases, such as database buffering, cache coherence schemes in general, and virtual memory. These related approaches are compared to ViFuR in Sections 8.3.1 and 8.4. A key difference of ViFuR-A compared to such approaches is that other replicas are available only at peer nodes, whereas, in database buffering, other approaches typically have a complete backup or complete secondary storage available. ViFuR-A needs to maintain the location of other replicas, and other replicas are typically scattered over nodes in the system. ViFuR schemes require local availability of object replicas for transaction timeliness. Both buffering and caching approaches are used to improve performance by trying to avoid network traffic, while ViFuR schemes disallow data access at other nodes entirely. Buffering and caching are approaches
ViFuR-A is a scheme that adaptively changes the allocation to meet access patterns that cannot be pre-specified. It is a scheme with an adaptation and setup time for local replicas, suited for soft real-time applications that tolerate that the first access to a data object may be delayed, while subsequent transactions execute timely. ViFuR-A complements ViFuR-S such that segments can be first setup off-line based on a pre-specification of known accesses, and then adaptively changed on-line. Replicas may be defined for permanent allocation to a node, by pinning the replica to the node (Section 5.1.2). A pinned replica can never be removed from the node to which it is pinned. For replicas allocated to nodes off-line, pinning ensures that such replicas will stay at a node. When no pre-specification is available, ViFuR-A starts with a database where data is not replicated, that is, segments are empty. Accesses detected on-line will incrementally change segments. Off-line specification and pinning allow ViFuR-A to support both hard and soft real-time transactions in an adaptive system, where there is no setup time for objects accessed by hard real-time transactions. Pinned replicas will have the same resource usage as any object has in ViFuR-S.

6.2.1 Adaptive segment management
As in ViFuR-S, the segments at each node contain the objects only required at the local node, and other objects, not allocated at the nodes, are unknown at the node. With ViFuR-A, all local objects are contained in segments at the local node in the same way as with ViFuR-S. This chapter focuses on segmentation for allocation, without the loss of generality of using multiple properties. As with static segmentation of ViFuR-S, segmentation for node allocation is based on object accesses, and this is specified by the transaction accessing an object. In ViFuR-A, segments can be changed during execution time (on-line) by adaptation segmentation, using the information being obtained from the transaction currently accessing objects at a node.

With the direct on-line adaptation of segments, there is no need for a matrix of objects, nodes, and object properties, such as used for off-line
input : add object at node
segment = getSegment(object);
if isAtNode(object) then
    newSettings = segment.getSettings();
end
newSettings.add(node);
/*Search for a segment with these settings*/
i=0;
while i < segments.size() and findSeg == null do
    testSeg = segments.get(i);
    testSettings = testSeg.getSettings();
    if testSettings == newSettings then
        findSeg = testSeg;
    end
    i++;
end
/*If not found, create new segment*/
if findseg == null then
    findSeg = segments.createSegment(newSettings);
end
setSegment(object,findSeg);

Algorithm 6.1: add() operation for incremental segment change

preparation of segments. Instead, adaptation incrementally updates segments to the current data need on-line. Segments are changed incrementally at each node locally, by using two operations: add("object at node") and remove("object at node"), which adds and removes objects to segments that have different property settings. These operations are allowed at the local node only and are used for local objects. Changes then propagate to other replicas of the same segment. The operations used execute atomically. They also commute, such that they are order-independent at the local node, to incrementally change the local segments. Both operations execute in $O(s + o)$, where $s$ is the number of segments and $o$ is the number of objects. The add() and remove() operations work on single objects or a set of objects having the same properties. In order to simplify the presentation,
6.2 ViFuR-A

```
input : remove object at node
if isAtNode(object) then
    segment = getSegment(object);
    newSettings = segment.getSettings();
    if newSettings.contains(node) then
        newSettings.remove(node);
    end
/*Search for a segment with these settings */
i=0;
while i < segments.size() and found == false do
    testSeg = segments.get(i);
    testSettings = testSeg.getSettings();
    if testSettings == newSettings then
        findSeg = testSeg;
        found = true;
    end
    i++;
end
/*If not found, create new segment */
if findseg == null then
    findSeg = segments.createSegment(newSettings);
end
setSegment(object,findSeg);
/*Remove segments with no info about local objects */
i=0;
while i < segments.size() do
    testSeg = segments.get(i);
    if testSeg.settings.contains(LOCAL_NODE) == false then
        segments.remove(testSeg);
    end
    i++;
end
end
```

Algorithm 6.2: remove() operation for incremental segment change

we use single objects for the presentation of the add and remove operations below.
Each segment, \( s_j = \{\{p_j\}, \{o_j\}\} \), contains two sets, the list of properties, \( p_j \), and the list of objects, \( o_j \). For a segmentation on allocation, the list of properties contains the list of current nodes, to allocate the segment objects. The add() operation (Algorithm 6.1) first searches existing segments for a set of nodes (and other properties, if multiple properties are used), which contain the wanted allocation for the object to add. If such a segment exists, the object is joined to it. If it does not exist, a segment is created containing only the new object. The remove() operation (Algorithm 6.2) first searches existing segments for one that uses the current node set of the object, but where the node of removal is not included. If such a segment is found on the node, the object is assigned to that segment instead of the current segment. If such segment is not found, a new segment is created with the wanted node set. Finally, segments with no objects at the local node are removed from the node.

Incremental changes to segments, using add() and remove() operations, are always initiated for an allocation at the local node, and then propagated to other nodes with replicas of the object that is changed. When received at another node, that node applies the same operation to its segments. Eventually, all the replicas have received the operation and applied it, and then the new replica is fully established or removed.

The storage of the segments at a node is bounded in size to \( O(o + \sum_{i=1}^{p} |P_i|) \), where \( p \) is the number of properties used, and \( |P_i| \) is the cardinality of each property. For the non-mutual property of allocation, \( |P_i| \) is the number of nodes where the segment is stored.

### 6.2.2 Object directory

Segment and property information is stored at each node for the objects currently allocated to that node. To establish other objects at the node, we use a replicated object directory (or simply 'directory') to lookup allocations of other data objects. This directory is a name service that stores the current allocation of all database objects. In addition, the directory holds global information about all the objects. The directory is replicated to a bounded number of nodes, \( d \). With such a bound, the storage used for a replicated directory scales with the degree of replication of the directory \( d \).
Any data object replica can be quickly found by a lookup at a directory node. It is only the property of allocation that needs to be stored in the directory. Other properties in use can be exchanged directly between peer database nodes, once the location of other replicas have been returned from the directory.

The storage in a directory is segmented the same way as in peer database nodes, and segments are updated incrementally by add() and remove() operations, received from database nodes that have changed their object allocations. Using a directory to locate other replicas during adaptation requires network communication with the directory during the adaptation, which makes the adaptation dependent on having a connection with at least one directory node. In order to add objects to an existing segment at a local node (adaptation type 1) there is no need for network communication. However, as soon as replicas are fetched from other nodes (adaptation types 2 and 3) adaptations require directory communication. Even without such directory communication, there will be communication for setting up replicas from other nodes of missing objects. Further, the added directory communication does not dominate the total data replica setup time, since the amount of data be transferred for directory communication is much smaller than the transfer of data objects during establishment. Directory nodes are allocated to database nodes that have high availability and low communication cost for as many database nodes as possible. Dynamic reconfiguration of directory nodes may be needed if connectivity or link quality changes, but we leave these issues for future work. At every database node there is also a directory list of the nodes with a directory. This list is used to address directory nodes for lookups. The directory list is bounded in size, since there is bounded number of directory nodes in the system.

An alternative to a name service, such as a directory node, would be to search for object replicas at all the database nodes in the system. However, with a directory service at a few nodes, the lookup cost is much lower. A replica establishment communicates with one node, while a search approach needs to communicate with all other database nodes in a worst case scenario. In systems where a directory cannot be used, data must be found in other ways. A simple search approach can be complemented with
Adaptive Segmentation for Dynamic Object Allocations

a distributed algorithm for the keying of data locations. In peer-to-peer systems, data is often found by using Distributed Hash Tables (DHT), such as in the OceanStore (Kubiatowicz, Bindel, Chen, Czerwinski, Eaton, Geels, Gummadi, Rhea, Weatherspoon, Wells & Zhao 2000), CAN (Ratnasamy, Francis, Handley, Karp & Schenker 2001), Pastry (Rowstron & Druschel 2001), or Chord data lookup systems (Stoica, Morris, Liben-Nowell, Karger, Kaashoek, Dabek & Balakrishnan 2003). In Chord, a distributed name service uses a distributed hash table for key-node mapping of data and for lookups of locations. We leave the usage of a DHT based approach in ViFuR-A for future work.

6.2.3 Specification of data requirements

The current set of replicas that needs to be available at a node is determined by the union of the current working sets of the transaction executing at that node, and we call this the access set at a node, \( A \), where \( A = \bigcup_{i=1}^{t} W^i S_i \). When transactions begin that need to use objects not available at the node of execution, a data fault is raised, which initiates an establishment of missing objects. If a transaction’s working set changes due to a mode change, or if the set of transactions changes at a node, there is an adaptation point that may need to change the set of local replicas. At each adaptation point, a recovery set of replicas \( O_r \) needs to be added to \( A \), and a replacement set \( O_p \) (which may be an empty set) needs to be removed from \( A \). The new access set will become \( A' = (A \setminus O_p) \cup O_r \). The storage available at a node (the buffer size) \( B \) must be greater than the access set, \( |A| \leq B \), otherwise thrashing will occur. Thrashing can also result from establishment and removals of the same objects at a high frequency, and may occur when the removal policy is not suited for the access pattern and access frequencies. The removal of objects should not be done for accesses that can be expected to be repeated within a short time, since there is a waiting and resource cost for the setup of replicas. In this section, we present a generic removal policy, which is suitable when access patterns are unknown and takes the utility of keeping a replica into account.

With data fault ratio we mean the ratio \( O_r / A \) of objects that change at an adaptation point. The adaptation interval \( t_{int} \) is the period between
adaptation points, and the adaptation time $t_a$ is the time needed to setup the new access set $\mathcal{A}'$ at an adaptation point, which depends on the write sets of the executing transactions.

### 6.2.4 Replica establishment

Along with the incremental change of the segments representing data allocations, as described in Section 6.2.1, the data objects themselves need to be setup and removed at nodes. We call the total setup process an establishment of a replica. This includes adding objects to segments, both at database nodes and directories, loading a consistent replica to a new node, and finally informing all nodes with a replica of the object about the new allocation. We call the similar process of de-allocating a replica, the removal of the replica.

The on-demand establishment of replicas allows any transaction program of a database application to be instantiated at any node of the distributed database. There is a cost for an adaptation process that makes local data replicas available on demand, in the delay of the first local access of a replica. Such delay includes: the cost of finding a replica, loading it to the local node, and updating segments at other nodes that have replicas. The adaptation cost of finding replicas and loading them can be reduced by designing the network appropriately, with short paths to directory nodes, and high bandwidth where needed. A transaction that accesses an object, which does not have a local replica, is blocked until a local replica has been setup. This is done by an adaptation protocol that requests the current allocations of the missing replica or replicas from one of the directory nodes. If such object does not exist in the logical database as described by the directory, a first replica of a new object is created at the local database node. If a replica does exist at some node, it is loaded onto the local node of the transaction. ViFuR uses incremental recovery (Section 2.6) to load a replica of an object (or a consistent set of replicas) from another database node, without locking the database at the other database node (Andler et al. 1999). The recovery source is the database node holding the existing replica, while the recovery target is the database node that will receive a copy of the existing replica. The most suitable (typically the closest) database node, as listed by the directory, is selected to be the recovery source.
Protocol

The replica setup protocol timeline is shown in Figures 6.1 and 6.2, and the protocol is outlined as: 1) Check local availability of objects required by the transaction. 2) If an object is missing locally, request the directory for the current allocations of the object. 2a) If the directory has the missing object, the directory replies with the current allocations for it. The recovery target, where a transaction is blocked waiting for a replica, sends a recovery request to a selected node with a current allocation of the missing object. The object or set of missing objects are loaded from that node by incremental recovery. It should be noted that the missing objects of a transaction (or even a set of concurrently executing transactions) may reside at different nodes. Objects to be recovered from the same node form a group that can be recovered as one recovery block. The more objects to be recovered from a node, the less the protocol overhead will be for each recovered object. As soon as a new replica is recovered, it can be used at the node (Figure 6.1). 2b) If an object is not available in the directory, a new object ID is reserved in it. The object is then created at the database node of the transaction. As soon as a new object has been created, it can be used at the node (Figure 6.2) and the transaction may continue. 3) The node reports the new existence of the object to the directory.

Cost of replica establishment

At an adaptation point, when there is a change of the objects accessed at a node, the node asks the directory for the allocation of $|O_c|$ objects. The network load for such a request, $l_{req}$ is $l_{req} = |O_c| + |O_c| \times k$, where $k$ is the bound for the degree of replication of objects. For the recovery of the missing replicas, the network load is $l_{rec} = |O_c| \times |o_{max}|$ bytes, where $|o_{max}|$ is the maximum size of an object. To update the directory and other nodes of the new replica, the update load is $l_{updc} = |O_c| + |O_c| \times ((k - 1) + (k_d - 1))$, since $|O_c|$ bytes are first sent to the directory and then the directory updates all other nodes, using $|O_c| \times (k - 1)$ bytes, and all other directory nodes, using $|O_c| \times (k_d - 1)$ bytes. Similarly, for replaced objects, the replacement load is $l_{upd} = |O_p| + |O_p| \times ((k - 1) + (k_d - 1))$ bytes. The worst case load,
where all missing objects $O_c$ are recovered from another node, and where $O_p$ is replaced, will be $l_{max} = l_{req} + l_{rec} + l_{updc} + l_{updp}$ bytes. The adaptation time for recovery, $t_a$, will be $t_a = t_s + t_t + t_d + l_{max}/H$, where $H$ is the available bandwidth, and $t_s, t_t$ and $t_d$ are processing times at the source, target and directory nodes respectively. With $n$ nodes adapting concurrently at a shared network, the overall adaptation time is $T_a = t_a \times n$. With sufficient computing resources at the nodes, the size of the change and the bandwidth of the network contribute most to the delay.

![Figure 6.1: Establishment of a new replica of an existing object](image)

The recovery source needs to act as a representative for existing replicas, a proxy, during the entire establishment of a replica. Updates to any current replica during establishment need to be sent to the new replica. As long as the replica has not been fully established, any node may send updates to the object without the updater knowing about the new replica. This is
why the recovery source forwards all updates during the establishment, and
the recovery source is the last node to receive the add() operation, such that
the forwarding of updates does not cease until all the nodes know about the
new replica. For this description of the protocol, the underlying network
is assumed to guarantee the order of delivery. For networks with differen-
tiated links and where order cannot be guaranteed, the protocol needs to
be extended, such that each message to nodes with existing replicas about
a new allocation needs to be acknowledged back. This will ensure that
update forwarding continues until all the nodes but the recovery source have
received their updates of the segments.

6.2.5 Replica removal

Similar to the establishment of a replica at a node, the removal of a replica is
initiated at the local node. It can be initiated by a removal policy, an explicit
message from a user transaction, or by a request from the directory. With
the working sets of transactions as the only specification of the data needs,
there are no explicit removal messages from the application. Instead, there
is removal functionality at the node to decide which replicas should remain
and which ones can be removed. ViFuR-A uses a generic removal policy to
ensure that replicas with high rate of accesses are favored at the node. There
are several such removal policy schemes available in the areas of database
buffering, cache coherence, and virtual memory. Many of these schemes
are based on counting and measuring references to data objects, and such
schemes are meant to model the load in a way that supports access pat-
terns. Such patterns are application-specific, which is why we use a gener-
ic removal policy for our generic ViFuR-A scheme. We also assume that
access patterns are unknown or random. More specific removal policies are
applicable when the types of access patterns are known. The general policy
used in ViFuR-A measures the average of references to a replica over time
to avoid the removal of replicas that are likely to be referenced again soon.
A good policy will avoid oscillation (or thrashing) between the establish-
ments and removals of allocations. Thrashing occurs when the processing
of adaptations takes so much time that no processing time remains for the
system to do any useful work. Thrashing may also occur when the storage
6.2 ViFuR-A

at the node is less than the amount of storage needed to hold the currently needed local replicas. Instead, the available local storage must be larger than what a certain replacement policy requires under a certain given workload.

Protocol

The removal protocol time line is shown in Figure 6.3, and the protocol is outlined as: 1) The removal policy (Section 6.2.7) detects that an object is not used anymore, and may be removed. 2) Since the directory ensures the minimum degree of replication, it must be asked first before a removal is allowed at the local node. 3) The local set of removal prevention rules (Section 6.2.6) must be passed to allow removals of the replica. The processing of local rules with the actual removal at the node need to be processed as an atomic operation. The reason is that concurrent removal initiations are possible, due to possible delays caused by the network communication. 4) The local replica is removed if allowed both by the directory and the local rules. Then the other nodes with replicas of the same object are informed about the removal through a directory message.

After a removal, the allocation information at other replicas is stale. Before the removal has been received at nodes with replicas, other nodes may send updates for the object at the node from where it was just removed. Such updates are ignored at the node of removal since there is no replica left to update. Such updates based on stale information will only be sent by
other nodes temporarily, and will cease once all other replicas have received the removal notification from the directory.

6.2.6 Removal preventions

Some replicas may be in use, even if they have not been accessed by transactions for some time. There are several reasons that prevent a removal, which are caused by the replica still being needed:

1. The directory must disallow the removal of an object when the replica is the last replica of the data object in the distributed database. In addition, a certain minimum degree of replication of objects may be guaranteed for the fault tolerance of the database application, and the directory is responsible for not allowing less than this minimum degree of replication. This rule is a global removal prevention rule, while other removal prevention rules are used at the local node where removal is initiated.

2. If a replica is in use as a recovery source node in the ongoing establishment of a new replica, the replica cannot be removed as long as the establishment is in process. The replica must remain at the node until the establishment is completed at the target node.

3. A replica cannot be removed if there is a blocked transaction that is going to use the replica when it is unblocked. A transaction may be blocked, waiting for another replica in its working set to be established. Until the transaction is released for execution, all replicas in the working set must be protected from removal.

4. Even when all the replicas for a transaction have been established, a transaction may not start to use the established objects immediately. The transaction may be delayed by some other processing at the node (typically the processing of other transactions). Replicas must be protected from removal from the point at which replicas are established until objects are in use and locked by an executing transaction.
5. Objects being used by executing transactions are locked and cannot be removed.

6. *Pinned* replicas are explicitly protected from removal.

For removal prevention rule 1, system-wide constraints are considered, and the directory is needed to approve such removals. A directory therefore needs to be available to the node of the removal. Due to removal prevention rules, removals may be delayed for some time even if replicas are not used, and such delays will temporarily use more resources than what is optimal for the actual current need. However, such non-optimal usage due to temporarily stored extra object replicas is the price paid to ensure availability of local replicas, to support timely execution of transactions. Transaction timeliness, ensured by the local availability of data, is higher prioritized for a real-time database than always having an optimal usage of resources.

6.2.7 Removal decision policy

The decision when to remove a replica is essential for resource management. It is important to remove a replica that is no longer being used by any transaction, since such replicas will continue to receive updates from other replicas using resources, and these resources will thus be used for no purpose. Since ViFuR achieves scalability by reducing and bounding the number of replicas, keeping replicas no longer in use will reduce scalability. Unused replicas need to be removed as soon as possible after accesses cease. However, high availability is important for the timeliness of transaction execution, and if accesses are expected to recur within a short time, there is a value in keeping the replica. Thus, a trade-off decision needs to be taken potential removal is detected, whether to keep the replica for high availability or reduce the number of replicas to favor scalability. We approach this tradeoff by estimating a *utility* for keeping/removing a replica. The decision to remove an unused replica is taken at each local database node, and when accesses cease, the replica must be removed unless there is a high utility for retaining it.

For many caching schemes, the keep/remove decision is often approached by counting past references to objects in order to estimate expected future
accesses. With regard to caching schemes in general, there is no high cost for keeping a cached copy as long as the local buffer is not full. For cache coherence schemes specifically, there is a small cost in processing and a communication cost for having a cached copy, since consistency between the cache copy and the next level of storage must be ensured. However, with a ViFuR database, there are two main differences from traditional database buffering and cache coherence schemes: 1) All transactions execute locally on local replicas of database objects, and there is no secondary database storage to access. The transaction is blocked until local replicas of objects become available. 2) Removal of replicas from the local storage is beneficial for scalability and for reducing resource usage. There is also a relatively high cost for updating replicas being kept. The high cost of keeping replicas requires that ViFuR applies a more aggressive removal strategy than than the one a caching scheme needs to use.

Inspired by the BYHR database buffering scheme, we measure an the average access rate of a replica during the period of allocation (Malik, Burns & Chaudhary 2005). The average access rate is a good estimator of future accesses for a usage period when the access patterns are unknown (Malik et al. 2005). More advanced estimations are useful when access patterns are known or can be predicted. However, we regard advanced pattern prediction and on-line access pattern analysis as future work.

For our generic model, we divide accesses into active and non-active periods of time. We call a time period of frequent accesses to a replica an A-episode, which alternates with a period of no accesses that we call an NA-episode. The shift between periods is atomic and triggered by specific events. We can detect the shift from A-episode to NA-episode by comparing the current measured rate of accesses with the average rate of accesses in the current A-episode. The shift from NA-episode to A-episode is detected by the first access to the object during a NA-episode. In order to measure the average rate of accesses, we define the rate-estimate, $RE$, which is the average rate of accesses during the A-episode (Equation 6.1, adapted from (Malik et al. 2005)), where an object, $o_i$, is accessed, $y_{i,j}$, by the transaction $T_j$, during the time $(t - t_0)$.
6.2 ViFuR-A

\[ RE_i(t) = \sum_{j} y_{i,j} \frac{(t - t_0)}{(t - t_0)} \] (6.1)

The decision to remove is based on a detection of the end of an A-episode for a replica (Figure 6.4). \( RE \) is used for detecting the end of an A-episode, and when accesses no longer arrive at the expected average rate, an NA-episode starts. The decision to remove is based on \( RE \) compared to the time passed since the latest access. For each object allocated to the node, three metric variables are needed: 1) The latest time of access (reference time for latest access) 2) The time of the start of the A-episode 3) The count of accesses since the start of the A-episode. If an access appears before the \( RE \) period ends, the reference time is updated. If one \( RE \) period expires without a new access to the replica, the A-period may potentially end. If an access appears before the second \( RE \) period expires, the reference time is updated. After two \( RE \) periods expire, the NA-episode begins, and the replica becomes a candidate for removal.

![Figure 6.4: Episodes and pattern detection](image)

The removal decision is complemented with a keep decision for candidates of removal. There may be a value in keeping a candidate from removal, since there is a cost associated with re-establishing a removed replica when new accesses re-appear. The replica is removed only if the cost of the continued receiving of updates exceeds the cost of establishing it again. The utility of the replica is estimated, as the ratio between 1) the advantage of keeping the replica at the cost of receiving updates from other nodes during a period known from the average frequency of past accesses, and 2) the cost of re-establishing the replica at the node. For the
removal/keep decisions, two types of $RE$ are calculated for each replica: 1) $RE_{local}$, which is based on the rate of accesses from local user transactions and 2) $RE_{remote}$, which is based on updates received from other nodes.

During the NA-episode, the utility of keeping the replica is periodically re-evaluated. As soon as the cost of keeping the replica is higher than the cost of re-establishing it, the replica will be removed. A good estimate of the resources required for keeping or removing a replica is the bandwidth used. For the updates received by the replica, we assume that the storage capacity on the node is enough to store the replica, and that the delay depends on the amount of data to be sent on the network to establish a replica. Let $m_i$ be message size in bytes used to update object $o_i$ from a remote node, and let $M_i$ be the message size in bytes to use at the network to recover object $o_i$. The utility $U$ can be expressed as in Equation 6.2. With a utility, $U$, below 1, it is beneficial to keep the replica at the time of the start of the NA-episode.

$$U_i = \frac{RE_{remote} \cdot |m_i| \cdot (t - t_1)}{|M_i|}$$ (6.2)

6.2.8 Concurrency issues in segment adaptation

The adaptation protocol of ViFuR-A is distributed, and every adaptation request is initiated at the local node, where the need of data is recognized. The add() and remove() operations commute and are order-independent, both at the local node and the directory, which ensures that the segmentation information eventually becomes consistent. However, some interleaving of these operations may still result in inconsistent segments, due to concurrency that needs to be considered. In the following situations, concurrency of adaptations is intricate:

1. Assume there are six nodes, N1-N6. N1 acts as recovery source for recovery target N3, while N2 acts as recovery source for recovery target N4. Assume that object $o_i$ is recovered concurrently to nodes N3 and N4, from nodes N1 and N2 (Figure 6.5). During the setup of the new replicas of $o_i$, the recovery sources propagate any object data updates to the recovery targets. Due to different recovery sources, different updates for the same object $o_i$ (e.g. from N5 and N6) may reach
each recovery target in a different orders, which could possibly cause inconsistencies between data replicas of the object ($o_i$ at $N3$ and $o_i$ at $N4$). However, a conflict detection and resolution mechanism (such as the PRiDe protocol in DeeDS) will detect and resolve such conflicts.

**Figure 6.5: Parallel recovery with updates of the same object**

**Figure 6.6: Concurrent recovery of an object to multiple nodes**

2. An object, $o_1$ is concurrently recovered from the same recovery source for two different recovery targets, $N1$ and $N2$. The directory, $D$, distributes new allocations and removals to existing replicas, using infor-
Adaptive Segmentation for Dynamic Object Allocations

mation from the recovery target at the time of recovery completion (Figure 6.6). The recovery at $N_2$ of object $o_1$ will not know, at recovery completion, that $N_1$ will need the new allocation when the recovery at $N_1$ is completed somewhat later. With this concurrency problem, an extension for the directory processing is needed. A remedy is to list all concurrently ongoing recoveries of the same object from the directory. At recovery completion, the content of the list is used to augment the message sent to all other nodes with a replica. In the example, this list will contain "$o_1$ at $N_2$" at recovery completion for $N_1$. With this information, $N_1$ will know about the new allocation "$o_1$ at $N_2$". For each additional concurrent recovery, the list will increase with one entry per object and node.

6.3 Evaluation of ViFuR-A

For the performance and scalability evaluation of ViFuR-A, we simulate a virtually fully replicated distributed real-time database with adaptive segmentation, as described in Section 6.2. We chose a simulation-based evaluation of ViFuR-A, since a simulation enables an evaluation of a large number of database nodes. An actual hardware setup for a system of the size we evaluated is not practical, and may not even be feasible to maintain in a lab setting. The simulator is written in Java and runs on Java Virtual Machines from version 1.5. The simulator is based on code that has been used in other work, for evaluations of both centralized and distributed real-time databases (Kang, Son & Stankovic, 2002, Wei, Son, Stankovic & Kang 2003). For our work, the simulation has been extended to model the replication architecture of the DeeDS database and the ViFuR schemes. The simulation was executed on a Dell PowerEdge 2950 Blade Server with two Intel Xeon CPUs (each with four CPU cores running at 2.5 GHz) and 32 GB RAM, running Linux Ubuntu as the operating system and the IcedTea6/OpenJDK Java machine.

Our objectives for simulating a distributed real-time database are:

1. To measure the usage of three essential key resources, bandwidth, storage and processing delays, in a simulation of a distributed real-time database with eventual consistency that uses virtual full replication.
2. To detect how scale factors influence resource usage, by measuring usage of increasingly larger systems and increasingly larger loads for selected parameters, such as the number of transactions and their frequency, and an the increased demand for adaptation to changed data needs. In particular, we want to find parameter settings where resource usage grows at a non-scalable rate.

The validation of our simulation is two-fold. We first take a simulation that has been used by other authors to evaluate approaches for distributed real-time databases, and assume that their simulation models are valid (Kang et al. 2002) (Wei et al. 2003) (Wei, Aslinger, Son & Stankovic 2004). Secondly, we extend this simulation with features of the DeeDS prototype system, as mentioned at the beginning of Section 2.7. We evaluate the simulation model and the simulation settings using face validation, conceptual modeling and white-box validation, where a presentation and discussion in the research group is used to evaluate the simulation model. A few key configurations are examined for validity, by reasoning how these features are expected to affect behavior. We also compare these configurations with our analysis of the expected behavior of the system. The actual simulation implementation has been carefully tested by using control test cases, and thoroughly analyzing simulation results.

We present two sets of simulations in this section. The first set (S1) baselines the resource usage and delays of ViFuR-A with full and with no replication. The results show a large scalability advantage with ViFuR for dynamic loads with adaptation, reducing the usage of both storage and bandwidth compared to full replication. ViFuR approaches the low resource usage of no replication (while not suffering from network dependence for most of the accesses), while the delay for accessing new replicas is a magnitude lower than for no replication. The second set of simulations (S2) evaluates the scalability and adaptation cost of ViFuR-A, where we measure bandwidth, storage and delay time in a system with an increasing number of nodes and clients, as well as an increasing database size. In order to evaluate of cost of adaptation, bandwidth, storage and delay are measured for an increased number of change requests, modeled as the degree of the change. We also measure the period of the change, the hot-spots used by
database clients, as well as the effect of the size of the hot-spot. The simulation reveals that ViFuR-A maintains scalability of the database for a typical changing database workload with mode changes. In addition, bandwidth and storage usage are linear to the number of replicas (that is, constant for a bounded number of replicas).

We model the load for the distributed real-time database by placing clients at database nodes, where clients periodically issue transactions with a Poisson arrival (Gray 1991, Nicola & Jarke 2000) and use a hot-spot for the accesses to the database at the node to which it is assigned. The hot-spot accesses of each client are modeled as $b\cdot c$ accesses (Tay et al. 1985). These $b\cdot c$ accesses model hot-spot behavior through accessing a fraction, $c$, (also called regular granules) of the database objects by a share of all the transaction accesses, $b$. As a common model of hot-spot accesses, the values for $b$ and $c$ are often set to 80% and 20% respectively, so that 80% of the accesses go to 20% of the data (Wu 1993). Some authors use a lower value for $c$: 1%, 5% and 25% are used by Triantafillou and Taylor (Triantafillou & Taylor 1995), while Gray and Reuter (Gray & Reuter 1993) argue that $b$ and $c$ are typically more likely to be 99% and 1% respectively for many database applications. The $b\cdot c$ access model has also been generalized for multiple granules (Zhang & Hsu 1996). Often the ratio between write and read accesses, $r$, is chosen to be 20% - 25% (Wu 1993, Triantafillou & Taylor 1995).

For distributed real-time applications, hot sets of data are typically used for a period of time, and different periods need different hot sets. Such mode changes call for adaptation to take place at certain intervals, in order to adapt to new data needs. At such mode changes access patterns change, and local replicas need to be replaced. We model such hot-spots as hot-sets of database objects that are accessed frequently during a mode of operation. In this evaluation, we now continue to interchangeably use the terms hot-set and hot-spot for such set of hot objects, and use $c$ as the denomination of the hot-set.

Resource usage for different settings of $b$, $c$ is evaluated. In the evaluation of the cost of the change of the access set (the local replicas made available for the local hot-set need), a periodic change of the hot-spot is
used. The currently accessed objects of the database, $c_i'$, are a result of the current hot-spot in use at node $i$. The mode of operation of a database client changes over time and with such mode changes, the set of objects accessed by local database clients will change into another set of objects, $c_i''$. We evaluate the cost of adaptation for such changed data needs, by measuring the resource usage and delay during multiple changes to the hot-spot. This is modeled as a periodic change of the objects in $c_i$, at a change period, $t_{mode}$ that is individually set for each client (Poisson distribution) with the same average period. The new set of objects in $c_i''$ is randomly selected from the entire database. In addition, we also vary the degree of how much of the hot-spot is replaced. The database is accessed by typical transactions of a fixed size of 10 operations. As a standard parameter, we use 4 write operations, since this is close to the generically used model (Wu 1993). We also evaluate the usage of 1 to 10 write operations in steps. Database objects vary in size between 1-128 bytes, and a data object identifier has global name scope. The simulation of the network causes an end-to-end transmission delay, depending on the Ethernet packet size, where packages are delayed 50 microseconds for the minimum package size of 64 bytes, up to 1.2 milliseconds for packet sizes up to 1500 bytes. For larger data sizes, transmissions are divided into several packages, where the transmission delay is the sum of the delay of the packages.

We model a distributed real-time database, such as DeeDS (Section 2.2), which supports timeliness of local transactions by having all used replicas in main-memory. The ViFuR approach targets scalability by reducing the usage of database storage at the local node. ViFuR is particularly useful for main-memory databases by reducing the actual main-memory usage at the local node, since the available main-memory storage is typically much less than available disk storage. The local main-memory size needs to be large enough to match the actual needs of data at the local node, while not over-using resources. The buffer size needed is workload-dependent and needs to be a tuning parameter. The storage reduction from using ViFuR is connected to the workload, rather than to the available buffer size. Transactions execute at the local node even without buffer size tuning. However, timeliness may suffer from trashing, when the need for data changes too
frequently, or when more than the available buffer storage is required from the current workload. We design our evaluation to ensure that the local node main-memory is large enough for replicas requested by the workload specified. Also, we monitor CPU utilization to prevent that possible processor limitations interferes with adaptation and replication. With such an evaluation setup, we can study the scalability of ViFuR without suffering from the restrictions of the limitations of a certain application-related hardware setup.

Table 6.1 shows the standard parameters used in the simulation and the workload specification. For each simulation, a single or a few parameters are varied. We choose to use system parameters that can be expected in a typical scenario that makes use of a distributed real-time database for communication (such as a distributed embedded system where simple control processors exchange data). We also vary one or two parameters at a time, to measure the influence on resource usage and transaction delays. For the evaluation, a pinned replica of each data object is assigned as a first replica of each data object in the logical database, such that the objects in the database are evenly divided over the nodes. Additional replicas are set up on-demand, based on transactions the clients execute. Client transactions access the current hot-spot without considering what objects are already available at the node. Objects will be established as new replicas at the node, while others are already available as pinned replicas or previously needed ones. All removal requests are based on accesses from local transactions and are approved by the directory, according to the removal protocol in Section 6.2.5, such that a minimum degree of replication can be maintained. Each new database node added will contribute more pinned database objects linearly to the logical database, and the system is scaled to several hundred database nodes.

6.3.1 S1: Comparing to alternative replication strategies

We baseline ViFuR-A against two alternative approaches for replication, choosing full replication of data, where the entire database is locally available at every node, and no replication with distributed transactions executing only at the node of the original data allocation. Further, ViFuR using
6.3 Evaluation of ViFuR-A

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Database nodes</td>
<td>50</td>
</tr>
<tr>
<td>Database size</td>
<td>50 objects/node</td>
</tr>
<tr>
<td>Transaction size</td>
<td>10 transactions</td>
</tr>
<tr>
<td>Transaction update operations</td>
<td>4 of 10 operations</td>
</tr>
<tr>
<td>Data object size</td>
<td>(1-128) bytes (Uniform)</td>
</tr>
<tr>
<td>Network delay</td>
<td>0.05-1.2 ms</td>
</tr>
<tr>
<td>#Clients</td>
<td>10</td>
</tr>
<tr>
<td>Mode change period</td>
<td>10.0 sec (Poisson)</td>
</tr>
<tr>
<td>Hot-set</td>
<td>80% of time using 20 objects</td>
</tr>
<tr>
<td>Hot-spot keep-ratio</td>
<td>0%</td>
</tr>
<tr>
<td>Initial deallocation timeout</td>
<td>2.0 sec</td>
</tr>
<tr>
<td>Deallociation detection time</td>
<td>2 periods of the average access period measured</td>
</tr>
<tr>
<td>Simulated time</td>
<td>200 sec</td>
</tr>
<tr>
<td>Workload</td>
<td></td>
</tr>
<tr>
<td>Client transaction period</td>
<td>100 ms (Poisson arrival)</td>
</tr>
<tr>
<td>Client transaction size</td>
<td>10 objects/client</td>
</tr>
<tr>
<td>Simulation S1 Parameters</td>
<td>10 samples/measurement</td>
</tr>
<tr>
<td>#Clients</td>
<td>2, 5, 10, 20, 50</td>
</tr>
<tr>
<td>#Database nodes</td>
<td>10, 20, 30, 50, 100</td>
</tr>
<tr>
<td>Database size</td>
<td>20, 50, 100, 150, 200, 300</td>
</tr>
<tr>
<td>#Clients</td>
<td>2, 5, 10, 20, 50</td>
</tr>
<tr>
<td>Database size</td>
<td>20, 50, 100, 200, 500, 1000, 2000, 3000 objects/node</td>
</tr>
<tr>
<td>Mode change period</td>
<td>0.2, 0.4, 0.8, 1, 3, 5, 8, 10, 20 sec</td>
</tr>
<tr>
<td>Hot-set</td>
<td>80% of time using 10, 20, 50, 100, 200, 500 objects</td>
</tr>
<tr>
<td>Hot-set keep-ratio</td>
<td>0, 20, 50, 80%</td>
</tr>
<tr>
<td>Client transaction period</td>
<td>20, 30, 100, 200, 500 ms (Poisson arrival)</td>
</tr>
<tr>
<td>Transaction update operations</td>
<td>1, 2, 4, 8, 10 operations</td>
</tr>
</tbody>
</table>

Table 6.1: Simulation parameters

only static segmentation may be another alternative. However, for the flexible access of data, and the dynamic load intended, as defined in Table 6.1, with transactions accessing a hot-spot that changes over time, ViFuR-S effectively must allocate all objects at all nodes as with full replication. With a hot-spot model that randomly uses any data object of the hot-spot, all objects may potentially be accessed at some point in time. All objects must therefore be available at all nodes when using ViFuR-S, as with full replication, to allow such arbitrary accesses. We thus regard ViFuR-S and full replication to be equal for such a changing load, and compare ViFuR-A with full replication and no replication as alternative approaches for replication.

We cannot use any simple-majority or quorum-based replication strategies for baseline comparison, since assumptions and constraints differ from ViFuR. These kind of approaches aim to control consistency, such that it
can be guaranteed that the most recent or most valid update will always be used. Instead, ViFuR optimizes on the local availability of data for transaction timeliness. Partial replication schemes are not comparable in general, since they typically allocate replicas as a tradeoff between availability and communication cost. Specific partial replication schemes that allocate replicas to nodes of access could be a baseline. However, to the best of our knowledge, the schemes available use immediate consistency with distributed commit, which would not allow timely transactions independent of communication links. ViFuR may also be used with immediate consistency, as we also support in our generic architecture (Section 4.4). However, with immediate consistency using distributed commit transaction timeliness cannot be ensured.

With full replication, there is no waiting time for the establishment of replicas at the local node. Instead, full replication requires a significantly higher usage of resources for storage and for data object updates. With no replication, the resource usage for storage and updates to replicas is very low, since there is only one replica of each object. However, with no replication, an execution of a transaction must be done at a remote node, requiring a transfer of the transaction to that node, an execution of the transaction at the node, and then the return of the result, back to the node of the original execution. We model such remote execution by assuming that the entire set of objects for a transaction is available at a single remote node. A remote execution of the transaction thus includes resource usage for copying data to be written at the remote node, the execution of the transaction at the remote node, and the copying of data read remotely to be transferred back to the original node of execution. With all the objects used by transaction at the same remote node, we measure the best case for no replication processing, since the entire transaction can be transferred to a single remote node. Any other distribution of remote objects used by the transaction will use more resources, since the transaction must be split into several sub-transactions to be executed at different remote nodes.
6.3 Evaluation of ViFuR-A

S1-1: Increasing the number of clients

For the alternative replication approaches, we measure the total usage of storage, bandwidth over all nodes, and the total transaction delay time over all nodes during a simulated time of 120 seconds. With an increasing number of clients, there will be increasing requirements for replicas.

- For ViFuR-A, storage increases with the number of replicas, as the number of clients increase (Figure 6.7.1). When the number of clients approaches the number of nodes, the storage usage is about the same for FullRepl and ViFuR. With as many clients as the number of nodes, ViFuR uses somewhat more storage than FullRepl, for segment management. The figure clearly shows that a significant amount of memory can be saved in systems that require a low number of replicas.

- For ViFuR-A, bandwidth usage follows a similar pattern as the storage
Adaptive Segmentation for Dynamic Object Allocations

usage (Figure 6.7.2). With a low number of clients, resulting in a low number of replicas, the bandwidth usage is low. With the number of clients approaching the number of nodes, bandwidth usage approaches the bandwidth usage of FullRepl. No replication (‘NoRepl’) uses low bandwidth, since only the data objects used by clients will generate network messages when transferring transactions to the nodes for data, and the number of objects used are relatively low.

- For ViFuR-A, the delay for transactions when using ViFuR is a magnitude lower than with NoRepl. With ViFuR, only the first access to an object is delayed, while all accesses are delayed with NoRepl. FullRepl has no delay at all for transactions, since all data objects are available at all nodes.

S1-2: Increasing the number of nodes

---

Figure 6.8: S1-2: Storage, Bandwidth and Delay vs. #database nodes
6.3 Evaluation of ViFuR-A

We vary, in steps, the number of nodes from 10 to 100 nodes, and present resource usage in a log-log graph, for the total storage usage, bandwidth usage over all nodes, and the total transaction delay time over all nodes during the simulated time period. For each database node added, another 50 objects are added to the database, while the fixed number of clients continues to use the same transaction load. For FullRepl, the entire database is stored at every node, increasing the database at every database node with 50 objects for each node added. For NoRepl and ViFuR, each database node will store only the 50 primary replicas, and ViFuR also add replica as needed by clients. The total storage usage grows much faster with FullRepl than with both ViFuR and NoRepl. Since the number of clients is bounded in S1-2, the number of replicas will also be bounded. For ViFuR, the same number of additional replicas is stored, which explains the constant difference in storage between ViFuR and NoRepl when more nodes are added. There is no transaction delay for FullRepl, and the delay for NoRepl is almost a magnitude larger than ViFuR, while the delay for both grows somewhat with an increased number of nodes.

An important behavior of ViFuR when scaling up the number of nodes is that the usage growth for both storage and bandwidth is lower than with both FullRepl and NoRepl, while the total delay time flattens out for the number of nodes evaluated.

S1-3: Resource usage over time

The resource usage is constant over time for both full replication and for no replication, as can be expected, while for ViFuR replication, resource usage follows the work-load caused by changing the access sets of transactions. Even with a standard hot-spot of 80/20 the resource usage is kept low, and with a narrow hot-spot of 99/1, resource usage approaches NoRepl. The plot is based on 10 samples, which is why individual adaptation variations in resource usage and delays are smoothen out. Delays for no replication are very high and also vary a lot, due to the dynamic work load. The simulation shows that an adaptive algorithm for establishments and removals saves large amounts of storage and bandwidth, while reducing the delays when use with a typical work load.
6.3.2 S2: Scalability and adaptation

In simulation S2 we evaluate scalability of ViFuR-A by varying selected scale factors while measuring the storage, bandwidth, and the delay of adaptation.

In simulation S2-1 we measure resource usage when varying the number of nodes and number of clients. We evaluate a system with 20 to 300 nodes (in steps), combined with 2 to 50 clients (in steps) (Table 6.1). We measure resource usage for all combinations of these two scale factors.

In simulation S2-2, the size of the database is increased, while the number of nodes and clients is constant. The database is increased in size and evenly divided over the nodes, and all objects have at least a single replica. Additional object replicas are established on demand of the 10 fixed clients, based on the transactions executed. In such a setup, update replication uses only resources for data object replicas requested by clients.
6.3 Evaluation of ViFuR-A

In simulation S2-3, the period of hot-spot change is varied, while the amount of change and the number of nodes and clients remain constant. This evaluates if there are any oscillating effects, or other limiting behavior, from establishment and removals for the periods chosen.

In simulation S2-4, we evaluate the effect of the characteristics of the hot-spot, in terms of hot-spot size and the amount of change of the hot-spot in the adaptation. This simulation shows how the access characteristics of the application affect the resource usage, and indicates how important it is to know the application access characteristics when using ViFuR-A.

In simulation S2-5, we vary the type of work load by changing the share of update operations in transactions from 1 to 10, while also changing the transaction period. This evaluates how a different workload affects resource usage and delays. It also shows that bandwidth requirements may increase dramatically for short transaction periods, in particular with transactions that have a high share of write operations, and where there is a large amount of updates to replicate.

S2-1: Number of nodes and clients

The number of replicas increases with the number of clients concurrently using the same logical object, while the number of replicas does not increase for an increasing number of nodes accessed by a constant number of clients. In Figure 6.10 we see that both storage and bandwidth usage per node increase with the number of clients.

With more clients than nodes, the measurements (20 clients and 50 nodes) give an impression of an anomaly, in that the storage and bandwidth usage is almost the same as when the number of clients is equal to the number of nodes (20 clients and 20 nodes). Due to the relatively small number of objects at the node, multiple clients at the same node will reuse the replicas already present. Since the number of write operations increases with multiple update transactions to the same data object (20 clients and 50 nodes), the bandwidth usage is somewhat higher than with a single client at each node (20 clients and 20 nodes). The storage is however about the same.

Transaction delays are relatively constant as would be expected from
Figure 6.10: S2-1: Resource usage vs. #nodes and #clients

(1) Storage per node
(2) Bandwidth per node
(3) Average transaction delay

a single-hop, single-segment network that is used in our simulation. With more nodes, the recovery sources are more scattered, such that replica establishments need to access more nodes before a blocked transaction can run. There is a slight increase of the total delay with an increasing number of nodes.

S2-2: Database size

Increasing the database size only will store a growing number of pinned objects at a fixed number of nodes. With a fixed number of nodes, the data needs are not expected to change, and resource usage is expected to be linear to the number of replicas in use. Resource usage is expected to be constant for a constant number of clients with constant number of requests from them. We increase the database in steps from 10 to 2000 objects per
node, using a fixed set of 50 nodes and 10 clients.

The storage measured is the total storage divided over the number of nodes, including both the database itself, and the additional replicas setup by requests from clients. The usage of storage follows almost linearly the amount of storage added to increase the size of the database. Further, for smaller database sizes, the extra replicas established for the clients dominate, due to the small size of the database itself. This can be expected, since a database with fewer objects is replicated over a constant set of nodes. There will thus be a higher degree of replication for the objects that are available in the database. With larger databases, there are fewer replicas of each object.

Bandwidth usage follows the number of replicas needed. A larger number of replicas of each object will result in higher bandwidth usage, since more replicas will be established, and also be in use for update replication.
The delay for the establishments of new replicas is also affected by what dominates processing for different values on database size vs the number of requests for replicas. For very small databases, the delay is small for the transactions we use, and increases as the database grows. With larger databases the delay is almost constant, since the number of established replicas becomes a small minority compared to the size of the database.

S2-3: Hot-set change period

![Graph showing resource usage vs. mode change period](image)

Figure 6.12: S2-3: Resource usage vs. mode change period

The resource usage and adaptation delay are lower with less frequent changes to the hot-spots (longer change periods). This can be expected for adaptation in a homogenous network used here, in which the cost for establishing replicas from any other node is equal. The storage needed corresponds to the number of replicas. With a higher frequency of changes to the
hot-spot, a slightly higher number of replicas are stored, which is caused by the removal policy lagging behind the establishment of new replicas. The bandwidth usage follows the number of updates for the replicas currently stored, and also the cost of establishing new replicas required. Thus, the curve for bandwidth usage, for the shorter periods examined, is steeper than the curve for storage. Bandwidth usage is influenced by two factors, both the number of replicas and the processing of establishments, while storage is influenced by the currently allocated replicas.

The worst observed delays are measured as the single worst delay for each sample execution of 120 seconds of simulated time. This means that only 10 values of worst observed delay are stored for each measurement point in the simulation. This delay is measured in our evaluation as an indication of the range of outliers from the average delay. These observations cannot determine the worst-case delay of the scheme under the parameters defined for the simulation, but an indication only. The observed longest delays indicate that most long delays will be a few multiples of the average delay. This can be expected in a homogenous network where significant delays are mainly caused by network congestion, or from an overload of the directory. For our future work, we intend to explore conditions for bounds on establishment delays and to find bounds for establishment delays under loads specified in more detail. This would need more elaborate load and resource definitions, as well as algorithms to match resource requests and resource availability. Our current work is intended for loads in soft real-time databases where the load cannot be specified in such detail.

**S2-4: Hot-set size and replacement**

The size of the hot-set is closely connected with the characteristics and the type of the application. In order to examine the suitability of different types of applications, we vary the $b$ and $c$ parameters of the hot-set with settings that are commonly used in literature to model different database application characteristics, as introduced in Section 6.3. We also vary the degree of the change of the hot-set. A small hot-set keeps accessing a small set of local replicas, and need to establish few new replicas when the hot-set changes. With larger hot-sets, accesses are spread over more objects, and
more objects need to be established at changes of the hot-set. Further, the degree of change to the hot-set, specified here as a replacement ratio, affects the number of objects to be established during a change of the hot-set. The required change of the hot-set directly affects the bandwidth needed for the establishment of new replicas and the storage at the local node.

Simulation S2-4 uses different sizes of the hot-set used, and the size of the degree of change to the hot-set. In order to simulate mode changes of the application, the same hot-set is used for a period of time, $t_{mode}$, and then replaced by another. The hot-sets requested at different nodes are independent, which simulates a loosely connected distributed application sharing a database. Accesses to the hot-set are made by b-c accesses, where different values for b and c are used. We also use a non-spot access pattern, with both b and c set to 100. This is not a hot-set, but all objects may be accessed by all accesses equally, effectively allowing full flexibility of accesses to

Figure 6.13: S2-4: Resource usage vs. hot-set size and size of replacement
6.3 Evaluation of ViFuR-A

arbitrary objects at all nodes, as with full replication.

Figure 6.13 illustrates how both the b-c access pattern and the degree of hot-set change affect resource usage. For non-spot behavior and wide hot-sets, there is no significant difference in resource usage, both for storage and bandwidth. There is also not much difference in the establishment delays. However, we see that for narrow hot-sets, the resource usage is dramatically lower. A small number of objects are required, and thereby the usage of concurrent replicas is low, which influences resource usage. The standard value for the period of change to the hot-set is 5 seconds, which seems to be too low to affect the resource usage. The period is varied in S-3, where only the very shortest change periods seem to influence resource usage with any significance. For simulation S2-4, we find that the number of concurrently used replicas influence resource usage most, and that applications with these types of accesses may benefit from using ViFuR-A.

S2-5: Workload characteristic

To evaluate the effect of the type of transaction characteristics, we vary the share of write operations, while also varying the transaction period. Every update operation causes other replicas to be updated, such that bandwidth usage increases with a larger share of write operations in transactions. However, since transactions always access local objects, the share of write operations does not influence the amount of stored data at the local node.

We see that the storage is not linear to the periods of the transactions. This is explained by the generic deallocation policy being used not keeping up with deallocation with regard to the amount of establishments. This behavior is similar to that observed with moving clients who move at high speed over an area of nodes (Section 7.9). The difference is that while transactions stay at the node here and the hot-set changes, while the moving sinks in Section 7.3.4 accesses the same set of objects while moving between nodes.

The share of update operations influences the bandwidth usage, as expected. The bandwidth usage increases with shorter transaction periods. The same bandwidth usage can be observed with fast moving mobile clients, in Section 7.9. A high demand for changes of the local set of replicas seems
very costly much with regard to bandwidth usage, while the share of updates in transactions costs less. This is mainly due to the fact that update replication is done to a small set of other replicas. With a different workload, where the same objects have many replicas in the systems, a higher share of update operations would cost more in terms of bandwidth.

### 6.4 Conclusions

Comparing ViFuR-A with the alternatives of full replication and no replication (Section 6.3.1) clearly shows the advantages of adaptive virtual full replication. The difference in resource usage for a typical workload, compared to full replication, approaches the resource usage of no replication, while access delay is a fraction of the access delays of no replication.
6.4 Conclusions

Resource usage scales with several important scale factors. Scale factors that require more replicas when scaled up will use more storage and bandwidth. Nevertheless, the increase of resource usage is low.

6.4.1 Results, contributions and subproblems met

The scalability advantage is significant for ViFuR, at the cost of a small establishment delay, in particular compared to full replication. Figures 6.7 and 6.8 show that even with a client using a hot-set at every node, much less storage and bandwidth is used for the data needed. With regard to scaling up the system to many nodes, the growth of resource usage is lower than for no replication, at a scalable growth of delay time.

When more clients use data, more local replicas need to be established and updated, and resource usage increases linearly according to the number of replicas established. Resource usage is relatively constant for most of the mode change periods examined. Further, we see in Figure 6.12 that when the change period is close to the transaction period, storage and bandwidth resource usage increase, due to the rapidly changing data needs. Resource usage is typically linear to the size of the hot-set, and with larger hot-sets the number of local replicas increases (Figure 6.13). A higher ratio for updates will also use more bandwidth, and high transaction frequencies require more storage (Figure 6.14).

Establishment delay is relatively constant, but increases slightly with the number of nodes, mainly because the lookup function will need some more processing. With the homogenous network used in this evaluation, the communication cost is relatively constant. Usage in a heterogeneous network is evaluated in Chapter 7.

Our ViFuR-A approach for adaptive and incremental management of changes to segments solves subproblem 2a and 2b. A large-scale evaluation using simulation partly solves subproblem 2e (this subproblem is solved further in Chapter 7). Our directory-based replica management, using global segmentation for allocation, and communication protocols, solves subproblem 2c. Finally, our generic detection and de-allocation algorithm partly solves subproblem 2d (this subproblem is solved further in Chapter 7).
6.4.2 Discussion

The generic approach for de-allocation is chosen to be suitable for access patterns that have low complexity and can be specified with only two parameters. The initial timeout parameter ensures that replicas stay for a timeout period that is longer than the expected time for the next access in the active period. This parameter is sensitive to the time between periodic accesses. Further, the timeout for the de-allocation parameter defines the time an object should be kept for, as a multiple of the current average access rate. It also decides the longest jitter tolerated for the replica’s stay at the node. These parameters are related to the period (100 ms transaction period in our evaluation) and the jitter (transactions arrive with Poisson distribution in our evaluation) that can be expected.

The local buffer size is not bounded in our evaluation, since we choose to study ViFuR-A under resources that are sufficient for the workload. Limiting the resource availability may cause execution to reach breakdown conditions, resulting from limiting execution conditions. Such constraints need to be considered for an evaluation for specific application usage. For our evaluation of a generic approach we chose not to use such limitations.

Our evaluation uses a single directory node. Carrying out an evaluation with multiple and replicated directories, using a single directory node, may avoid any hot-spot limitations. We have neglected the synchronization issues related to using multiple directories, since we can assume that a distributed database with eventual consistency resolves any such consistency management between multiple directory nodes. Further, another reason for using a single directory node is that it more quickly reveals any synchronization errors caused by concurrency problems in our distributed protocol for replica establishments and removals.
Chapter 7

Using Virtual Full Replication

I have been impressed with the urgency of doing.
Knowing is not enough; we must apply.
Being willing is not enough; we must do.
- Leonardo da Vinci

The benefits of scalable resource usage in the utilization of Virtual Full Replication require that an actual application scenario will concurrently use a bounded subset of replicas of data object replicas in a distributed database. Many distributed applications have this behavior, and can make effective use of a distributed database as a middleware, in order to connect many users of data in a distributed application. A data-centric communication approach separates individual users of data in such an application, and reduces the requirements for coordination and synchronization.

We apply ViFuR-A in a typical large-scale resource-constrained scenario, which can potentially have a large number of nodes and updates, and a number of dynamic data users. In such a scenario, ViFuR adapts to reduce the number of replicas to no more than what is needed by the users, while ensuring availability of all data objects.
7.1 Using ViFuR in wireless sensor networks

A wireless sensor network (WSN) consists of a large number of communicating nodes, each one having a limited supply of energy. Sensor nodes (source nodes) are often deployed once, at fixed locations, but sensor locations may also change. Client nodes (sink nodes) use sensor data, as mobile nodes inside the network or as nodes at the network edge. The communication links are typically very limited, with low bandwidth and long communication delays. Communication protocols are often lossy, and sensor nodes have limited processor capacity.

The energy-efficient reporting of sensor information is a challenging task in WSNs. Approaches often include reducing the amount of communication and processing, as well as energy-efficient routing and routing changes due to adaptation to needs.

Wireless sensor networks (WSN) introduce new challenges for the research community in distributed real-time systems. WSNs are large-scale and resource-constrained applications, to which ViFuR brings a novel approach for scalable and flexible communication. It provides a multi-tiered communication approach with structured storage, and that also supports filtering and fusion of the data stored. ViFuR reduces communication needs, and thereby reduces resource and energy usage.

7.1.1 Issues with wireless sensor networks

A number of specific challenges need to be met for WSNs (D. Estrin & Kumar 1999, Stankovic, Abdelzaher, Lu, Sha & Hou 2003): 1) Sensors are usually battery operated, thus efficient use of the available energy supply is critical. Once the energy supply is depleted, the network must be recharged or abandoned. Transmissions require a relatively high energy cost, which means that delayed, coordinated and aggregated transmissions may save much energy and increase the lifetime of WSN nodes. 2) In comparison with local area networks, communication links in WSNs are several orders of magnitude slower. This gives rise to high latency, in particular for multi-hop transfers. 3) The connectivity is often unreliable, as communication paths may be lost or obstructed during operation, and the communication
range of sensor nodes is irregular and often dependent on battery level. 4) Further, due to power, size and cost limitations, the memory at each node is usually very limited.

WSNs are usually deployed as large-scale sensor applications monitoring some geographical area of interest. There are typically a few edge nodes that use high level sensor information, while mobile in-network nodes may use any of the information from sensors in the network. Usually there are only a few critical sensor events that must reach the network edge in bounded time, while less time-critical and possibly aggregated data may reach the network edges with best effort. A sensor node is most often the single updater of a specific sensor value, and the value is propagated through the network to be used by one or multiple clients.

Many propagation and routing protocols have been proposed to address the challenges within WSNs, to report sensor readings and critical timely events to the edges of the network. In order to reduce the amount of communication and processing in the network, data streams from sensors are often translated (or filtered) into application specific events, such as when a threshold value is reached. This is done by application specific programming of the sensor, or by using long running queries typically placed as close to the sensor as computing resources allow (Bonnet, Gehrke & Seshadri 2001), usually in the sensor. Queries may be declared at network edges and then distributed, such that the WSN returns sensor data as results filtered by these queries (Madden, Franklin, Hellerstein & Hong 2005). Several approaches assume that sensor locations are known, and that localization schemes are available. Consider a large surveillance or rescue mission, where a large number of sensors are deployed. Sensor nodes may form an infrastructure, but finding and communicating their data, in particular between locations far apart, involves many intermediate sensor nodes. Further, if many users of sensor data move through an area while using it, the routing of sensor data updates quickly, and becomes a very complex task to manage.
7.1.2 Comparing ViFuR for the LAN and WSN environments

In a previous chapter, we present and evaluate the usage of ViFuR in a local area network (LAN) environment, which offers high bandwidth and low latency. This environment allows a certain approach for ViFuR, in relying on the capacity of the network communication and that at least one directory node is available during adaptation. With WSNs, the execution environment is different in some critical aspects; the bandwidth is much lower and the connectivity is unreliable. The restrictions of the execution environment in a WSN, compared to a LAN environment, require some changes to the ViFuR-A approach to better suit the properties of the WSN environment. This is to both avoid critical problems and use opportunities of the characteristics of WSNs. This section compares the properties of the both the environments considered. The comparison motivates the changes to ViFuR-A, when used as a scheme suitable for WSNs.

In a LAN, nodes communicate using a single network segment, or they use a switched communication link over multiple network segments. The network segments, involved in a communication link, may be shared with other communication links. Also, each node with an IP address may communicate directly with another node with an IP address, using a connection-oriented link between them.

With WSNs there is a shared communication channel radio space that uses connection-less communication between nodes. Communication packages are sent over wireless channels and may collide with other nodes transmitting at the same time. There are routing protocols available for multi-hop communication over multiple hops, relaying communication over multiple WSN nodes. Such communication propagation paths are even more limited in bandwidth and reliability. For reliability, multi-hop communication paths need to be as short as possible.

Matching characteristics

Several characteristics for the usage of DeeDS with ViFuR-A in a LAN setting match characteristics of WSNs well. In Table 7.1, we list characteristics that we argue are similar or compatible, for using ViFuR-A in a LAN.
and using it in a WSN environment. A discussion about the characteristics follows.

<table>
<thead>
<tr>
<th>ViFuR-A on LAN</th>
<th>ViFuR-ASN on WSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded resource usage, Scalability</td>
<td>Limited resources, large scale applications that need to scale</td>
</tr>
<tr>
<td>Peer nodes / Tiered</td>
<td>Peer nodes (Tiered proposed)</td>
</tr>
<tr>
<td>Robust propagation</td>
<td>Unreliable connectivity</td>
</tr>
<tr>
<td>Detached replication</td>
<td>Slow links</td>
</tr>
<tr>
<td>Segments</td>
<td>Aggregations</td>
</tr>
<tr>
<td>Active rules</td>
<td>Long running queries</td>
</tr>
<tr>
<td>Adaptation for availability</td>
<td>Moving sinks</td>
</tr>
<tr>
<td>Replicas of data in segments</td>
<td>Fault tolerance needed</td>
</tr>
<tr>
<td>Whiteboard communication</td>
<td>Location related communication (end-to-end) can be complex to manage</td>
</tr>
</tbody>
</table>

Table 7.1: Matching characteristics

ViFuR-A reduces the resource usage based on the current data needs, while WSNs have limited resources, which is thus a matching characteristic. DeeDS database nodes using ViFuR are peer database nodes, just as typical WSN nodes. In both cases, nodes have a single purpose and performs a few key functions. The update propagation model of a DeeDS database is robust, using a store-and-forward replication approach, detached from the local updates of the database, such that propagation will function even when links are unreliable and nodes are temporarily partitioned from other nodes. WSNs have unreliable connectivity and slow links, making store-and-forward replication suitable, because propagation can be delayed to take place when the connection is available again.

Using segmentation, data can be divided at the local node, for local processing of data, and for remote propagation of data. This kind of split of the database supports aggregation in WSNs, such that there can be segments for the source node used for local processing such as aggregation, and other segments that are results of aggregation to be propagated to other nodes. Aggregated data is stored, typically in a more condensed form than the original data, in the second segment type, and is replicated to the set
Using Virtual Full Replication

of nodes using the aggregated data. Aggregations may be done in several steps, effectively building a hierarchical structure of segments with aggregated data. Such a hierarchy of aggregations allows lower-level information to be aggregated closer to the sensor, and then replicated to nodes as higher-level data where needed. Such *tiered* aggregation of information is suitable for Information Fusion scenarios, which may fuse data at a lower level in a WSN and then propagate fused data to upper nodes, for further usage or even higher-level fusion. In addition, active rules of the database allow the triggering of fusion algorithms and control of database updates.

Replication improves the reliability of a WSN, since both storage and processing may be distributed to multiple nodes, supporting fault tolerance such that nodes can crash but data will still be available in the system. Crashed nodes that had data replicas can be restored from other running nodes. After a node crash where replicas are lost, the degree of replication and fault tolerance can be restored again by using the adaptiveness of ViFuR-A. Finally, the whiteboard approach abstracts the complex many-to-many routing problem in a WSN with many data sources and multiple, dynamic and mobile data users.

Mismatching characteristics

Some characteristics of ViFuR-A, using a LAN, and those of WSNs do not match well. We list some important characteristics that we argue have conflicting characteristics. Knowing these characteristics enables us to present approaches to avoid or solve such conflicts. In Table 7.2 we list potential conflicts that need to be considered when using ViFuR in a WSN. There are approaches for possible solution for some of them, and a discussion about the characteristics and possible solutions follows.

An active database, such as the one we use, has active rules that are typically static. Many WSN implementations use static programs in each sensor, tailored for the data propagation of the application. Some data-centric approaches allow long-running queries that help to extract data at sensor nodes, and such queries can be updated during execution time. An active database would use a set of static rules to enable changed behavior when data filtering needs to change. Processing that is not stored as a part of the
ViFuR-A on LAN | ViFuR-ASN on WSN
---|---
Fixed active rules | Changing queries
Dynamic allocation of data, replication between replicas | Dynamic changes to queries and propagation paths
Full connectivity | Partial connectivity, but multi-hop routing available
Separation of database and communication layer | Small footprint required
Local timeliness | Global propagation delay timeliness
Multiple writer | Single writer
Propagation within the network | Propagation within and out of the network
Low latency, High bandwidth | High latency, Low bandwidth

Table 7.2: Mismatching characteristics

database can be triggered by active rules to execute arbitrary code, such that sensor data storage and propagation change, and such data processing may also communicate with the data user to change its behavior during execution. This allows updates to the filtering and aggregation processing, to be more flexible than with updated queries.

ViFuR-A adapts the allocation of data objects to the actual usage, while existing WSN approaches typically change the distributed queries or do not change at all.

There is full connectivity between nodes in a LAN, and all connections are typically low-latency high-bandwidth ones. A significant difference in a WSN is that sensors usually connect to a few other nodes, such that a message needs to pass multiple nodes to reach the edge of the network, using a multi-hop routing scheme.

The database and communication layers are separated in ViFuR-A, requiring both layers to be present at all the peer nodes. A WSN network is made up of limited capacity peer nodes that require a small footprint for the programs to run. Usage of the ViFuR-A approach in a WSN network clearly requires a different approach, since restricted WSN nodes may not have the capacity to execute a database node. More powerful future WSN hardware
cannot be expected to solve this entirely, since the requirement for a small footprint will remain in the future due to price and energy requirements of WSN nodes. Instead, a different paradigm is needed for scalable WSNs using a whiteboard approach for data distribution and propagation.

The detached replication approach that we use in ViFuR originates from real-time requirements on transaction execution. Any distributed application will benefit from short database access times so the replication approach is useful in WSNs as well. However, global timeliness of updates is hard to achieve without relying on predictable communication, both with or without a database approach. With the usage of wireless communication, the delays between nodes are even more unpredictable than with wired communication. One way of improving timeliness is to first keep the bandwidth usage at a low level compared to the capacity of the link. Reducing the bandwidth usage, as done in ViFuR-A, is one such approach. Then, use a higher capacity network for as many communication links as possible between the sensor and the user of the sensor data. This could involve using high-capacity IEEE 802.11 for key communication links, instead of IEEE 802.15.4 links that are typical in WSNs. Some key nodes may need to have access to more energy and other network hardware, for using IEEE 802.11.

A characteristic that differs in terms of the application is the number of writers to a logical data object. In a typical distributed database application with multiple replicas of the same logical object, there may be multiple writers to the same logical data object. Conflicts are typically rare (Syberfeldt 2007), but should they occur, conflict detection and resolution mechanisms need to be in place to solve them. As mentioned above, there is usually one updater to the information about one sensed value. Multiple sensors may detect the same entity in the environment, and such observations need to be associated and merged. However, the raw data value from each sensor is an individual reading of the environment. A straight-forward approach is to publish each sensor reading as an object in the database, and let aggregation and fusion algorithms handle the data association problem, at fusion nodes using the database. Each data object then has a single writer and a WSN application does not typically make use of a database conflict and resolution mechanism.
7.1 Using ViFuR in wireless sensor networks

A whiteboard approach exchanges data between the nodes connected to the database. A WSN application typically collects data from sensor nodes and propagates updates to the edge of the network. A database approach may support both data users inside the network, and nodes at its edge.

Other issues

A ViFuR scheme for WSN cannot rely on an available directory service to find data objects. Instead, it is necessary to search for objects. This is more suitable for WSNs, since the location of a sensor value is more important than the identification of the sensor data object. Location-based search approaches are common in data-centric approaches for WSN. It is common to make use of a localization scheme to provide location awareness in sensors. In addition, sensor data is typically tagged with both the locality of the sensor reading and also the sensor type (Ye, Luo, Cheng, Lu & Zhang 2002). Both types of meta-information about the sensor value can be used to search for sensor data in a certain region and for a certain sensor type, as done in Directed Diffusion (Intanagonwiwat, Govindan & Estrin 2000).

Readings from separate sensors can be stored as a separate data object, created at the database node to which the sensor is attached (Object ID is assigned there by means of a UUID that embeds the sensor location). A search for a data object (for example using a directed search by Directed Diffusion) returns the object ID, and can then be used for replica setup at the local node, as with ViFuR-A.

Local timeliness is of less concern in WSN. Instead, it is preferred that propagation time from source to sink is somewhat predictable. If not possible, at least the sensor reading needs to be tagged with a time-stamp to know the validity of the sensor value. The DeeDS database leaves propagation to the communication layer and queues updates for best effort propagation.

In WSNs, aggregation and filtering for bandwidth and energy preservation is central. Continuous queries and distributed filters are commonly used, preferably filtering can be dynamic. With a database approach this can be done by active queries triggered by updates to replicas. In DeeDS, active rules are statically assigned to nodes and executed there. There is no way of changing or moving a rule in case of changed data needs.
7.2 ViFuR-ASN

We apply ViFuR as a concrete case of ViFuR-A adapted for WSNs to make use of some specific key characteristics. In this section, the ViFuR-ASN scheme for using ViFuR in WSNs is presented. This is proposes as a scheme for timely access of sensor data and for the scalable propagation of updates across the wireless sensor network. We call this scheme (ViFuR-ASN). It enables large scale sensor networks and simplifies the addressing of data and nodes. Further, it provides in-network storage, data fusion and aggregation by active rules, and it supports fault tolerance by data replication. We show that ViFuR-ASN can meet the challenges of resource usage and scalability, of real-time requirements for local execution, and of the complexity of propagation from sensors to multiple mobile clients for networks with unreliable connectivity. A distributed real-time database with virtual full replication is used as a communication medium that allows a whiteboard approach for publishing and reading sensor data. Sensor values are made available to all clients at any location, by publishing sensor values as database objects to be used at any database node in the system. The ViFuR-ASN scheme is an application of virtual full replication with adaptive segmentation (ViFuR-A) for a WSN. This paper extends ViFuR-A by utilizing properties of wireless sensor networks, to find and allocate database object replicas where clients need them.

7.2.1 A database approach for communication in WSN

In ViFuR-ASN, sensor readings from sensor nodes are published as sensor data objects in a distributed real-time database, and sensor readings are accessed by database operations in transactions. We consider a wireless sensor network system of ten to hundreds of database nodes, possibly supporting from hundreds to several thousand wireless sensor nodes.

A two-tier approach is used that comprises a sensor tier and a database tier of nodes, where the database tier has more powerful nodes and greater energy supplies. We assume that both the sensor tier nodes and the database tier nodes have knowledge of their own locations, for example, by utilizing a localization scheme. Clients use sensor data as sensor data objects accessed
at database nodes, and a client can select which database node to currently use. Switching of database nodes is typically done when a client moves to another location in the network (Figure 7.1). Each database node has a virtually fully replicated database that stores the data objects for the sensor nodes connected to it, which is typically only a single or a few data objects for each sensor node.

Sensor nodes connect to a suitable (not always the physically closest) database node and update the corresponding sensor objects in the database. The set of sensor nodes that connects to the same database node makes up a sub-network of sensor nodes, which may connect to a database node over multiple hops in the sensor sub-network. The number of hops that a sensor node uses for its communication with the database node is bounded to the number of sensors in the sub-network. The actual number of hops from a sensor to a database node is typically lower, since sensor nodes are located at all sides of a database node.

In this solution, we adopt independent updates, virtual full replication,
conflict detection and conflict resolution approaches of DeeDS (Andler, Brohede, Gustavsson, & Mathiason 2007), for the timely execution of transactions and for network latency independence. We assume: 1) The network used by the database tier guarantees delivery and conceals multi-hop connections, so that the database nodes only perceive connections having a higher propagation delay. 2) Sensor sub-networks use a timely communication protocol. 3) The database tier may use either best effort or bounded time propagation communication. 4) Unique identifiers are assigned when new database nodes are added.

The infrastructure also provides fault tolerance by supporting the replication of the gateway node data on a bounded number of neighboring gateway nodes. Furthermore, this approach can be extended by letting each sensor connect to multiple gateway nodes. It will also reduce the need for neighbor database replication at the gateway level.

Using a distributed real-time database with virtual full replication in a wireless sensor network will make a local replica available where: 1) A sensor node updates a sensor object at a database node. This replica of the sensor object is typically the replica closest to the sensor, such that it will be the replica first updated by a sensor reading. Note, however, that the database architecture that we use does not distinguish replicas, but to the database system all replicas are primary replicas that can be updated independently. 2) A database node has a client connected that uses a sensor object. A client moving towards a new area where it is possible to connect to another database node, declares its data needs by executing a transaction with operations accessing data needed at the node. If a data object replica is missing at the node, the database node extends the set of available data replicas by establishing a new local replica. When the client moves beyond the reach of a previously accessed database node, the lack of accesses is detected by a deallocation algorithm, and the data object replica is removed.

7.2.2 Database tier

Each database node is a peer DeeDS database node, as described in Section 4.4.1. A database node can store a limited amount of data, but for our simulation evaluations we do not want to restrict measurements by this limiting
factor. Instead, we measure the maximum storage required for a selected set of workloads. Database nodes give unique identifiers to sensor objects. The location of each sensor is stored as a separate database object, and provides spatial context for sensor objects. Updates coming from sensor nodes will update sensor data objects, but they will not necessarily update the associated location object. Location objects are used by database nodes when searching for sensor information in a particular area of interest. Since all sensor locations are known at the database tier, a search for a sensor in an area of interest is done by searching only at the database nodes, without any need to (partially) flood sensor nodes.

**Replica updates**

Database replicas are updated by detached replication, which causes replicas of the sensor object to receive delayed updates. In a distributed system in general, a remote user of a sensor value will always use a delayed value. Independently of the replication approach used, a remote user of sensor updates will always need to wait for propagation of an update caused by the delay of the network. With our database approach the latest known update will always be available for read and writes without any delays. If the local usage conflicts with an update received later, this will be detected and resolved (Syberfeldt 2007). In general, communication in wireless sensor networks relies on unpredictable radio links, but several real-time approaches are available. With such real-time communication protocol in use, the update delay will be predictable under certain conditions.

**7.2.3 Sensor tier**

We organize sensor sub-networks to propagate updates over a few hops within each sub-network towards a nearby database node. Many approaches exist in the literature for ad-hoc sensor node organization, message routing, and sensor node localization, to organize networks of sensor nodes. For our approach, we assume that the sensor tier can organize sub-networks around database nodes, and that such sub-networks can timely deliver sensor updates to database nodes for bounded-sized sub-networks. Several
approaches exist for timely communication in WSN when there is a bound on the maximum number of communication hops. RT-Link (Rowe, Mangham & Rajkumar 2006) is an approach that allows end-to-end real-time guarantees for WSN. This is a TDMA-based approach that uses global clock synchronization to divide time between sensor nodes. The 802.15.4 communication standard, as a part of the Zigbee initiative, offers real-time communication when the number of sensor nodes in the WSN is known. Koubaa et al. (Koubaa, Cunha & Alves 2007) present an approach for time division communication that schedules beacon frames in a beacon enabled Zigbee communication setting. This approach allows more stringent timeliness requirements to be fulfilled for small sensor networks. Approaches such as SPEED (Krishnamurthy, He, Zhou, Stankovic & Son 2006) offer soft real-time communication by considering velocities of communication packages propagated through large WSNs.

7.2.4 Establishment and removal of replicas

It is possible to ask for any data object at any database node, by executing a transaction that accesses the object. If a local replica is missing, the transaction is blocked until a local replica has been established. The adaptation protocol of ViFuR-ASN consists of three components: 1) Find the data objects that need to be established. 2) Create a consistent replica of a set of database objects in use at another node. 3) Remove unused replicas that have no benefit.

Finding replicas

In order to establish a new replica, the nodes of existing replicas need to be known. For WSN applications, the interest for a certain area is a more important search type than a search for a certain data object with a known identifier. The ViFuR-A scheme (Chapter 6) is identifier-based and uses a replicated directory to list all current replicas and their identifiers. However, a directory is a critical resource that may be a single point of failure, or suffer from congestion. A directory is not needed in ViFuR-ASN, since data will always be available within a bounded search: 1) For every sensor
7.2 ViFuR-ASN

object in the database there is a known sensor location, and finding an area of interest is a matter of searching a bounded number of database nodes, instead of flooding an entire sensor network. Further, only those database nodes in the direction of the requested location are searched. 2) For moving clients, continuing to use a data set, the location based search has already been done once, and the data object identifiers are known. At a switch of the database, the newly connected node only needs a simple lookup by an identifier at neighboring nodes. Such lookups will typically find the data, one or a few hops away. With a typical topology this will be the immediate neighbors, or a neighbor only a few hops away, depending on the database node density and radio communication range.

The database node density and the degree of branching between database nodes will influence the number of database nodes being searched towards a location. In the worst case, the search will pass \(|N_i| - 1\) hops, where \(|N_i|\) is the overall number of database nodes. However, a search typically propagates as a search tree over the nodes. For each database node searched, \(b_i\) other nodes are searched, where \(b_i\) is the branch-out factor at database node \(i\). With a high \(b_i\), search concurrency increases and objects are found in fewer hops. A typical search in a search tree will need to search \(\log_b N\) nodes. For each database node, \(o_j\) sensor data objects will be searched, where \(|o_j|\) is the number of sensor data objects at database node \(i\), and where the search cost will be \(\log_2(o)\) for the set of sparse object identifiers sorted as a binary tree. The total search cost in hops for an object \(o_j\) is expressed in Equation 7.1.

\[
\text{cost}_j = \log_b(\log_2(o_j))
\] (7.1)

A moving client can choose different strategies for switching nodes. One strategy is to continue the accesses at a certain database node for as long as it is possible to remain connected to that node, and to switch to another database node once the connection is lost. Another strategy is to connect to the database node with the strongest radio signal. This causes more frequent adaptations and may cause switching to oscillate due to radio irregularity. Choosing which strategy to use is the client's decision, and different strategies affect resource usage differently. To avoid oscillation, we choose to switch database node only at a lost connection.
If a location-based search cannot find a replica, it means that there is a hole with no coverage in the WSN (a void), and the transaction cannot be executed. Resolutions to such a problem are application-dependent and may include using sensors at the borders of the void instead, or simply discarding the transaction.

The location of sensors may change, which causes the location information for the sensor data object in the database to be updated. Thus, a particular search that finds a sensor at a requested location may have a different location at a later update of the sensor value. For this reason the client needs to consider the location of the sensor that is currently being used and to possibly repeat the search.

**Adding a replica**

As with ViFuR-A (Section 6.2.4), the second step in establishing a new replica is to load a copy of an existing replica onto the node by incremental recovery. The most suitable (typically the closest) database node is selected to be the recovery source. Figure 7.2 shows the time line for the establishment of a replica, with database node \( N_r \) as recovery source and database node \( N_t \) as recovery target. The recovery messages are sent with best effort and the blocking time for a waiting transaction depends on both the number of hops searched and the time required for the incremental recovery, including the bandwidth available and the processing time available at the recovery source.

Following the principle that the transaction working set specifies the data needed at a node, a roaming protocol is not necessary for when a moving client switches database node and starts to use a replica at a new database node. No control messages need to be exchanged to coordinate the switch, not between the database and the database user, and not between two database nodes. Transactions and their read and write operations are sufficient for declaring the data need at each database node independently.

However, optimizations are possible. The client may choose to submit a "prospective" transaction in advance of actually needing data at a node. This lets the new node establish a new replica in advance of the actual first access. Such an optimization can be implemented for critical accesses and is
application-dependent. It will shorten the delay for the first access when the
database application knows its future nodes of access and future data needs.
These "prospective" allocations may introduce replicas that are not actually
used. However, this is a scalable behavior that only involves interactions
between a few database nodes, and excessive replicas will soon be removed.

![Image of replica setup]

**Figure 7.2: Setup of a new replica**

The transaction that needs to access a missing database object at a gate-
way, $G1$, is blocked until a local replica has been setup there. When a
replica of the sensor object has been set up locally, the blocked transaction
can proceed execution. If no other replica can be found, there is a hole with
no coverage (a void) in the network, and the required transaction cannot be
executed. Different application resolutions can be applied, such as waiting
at the borders of the void, or simply discarding the transaction. Transactions
are the only means of accessing a virtually fully replicated database, and no
additional control messages are exchanged at the interface of the database
and the database user. The protocol time line of the replica setup is shown
in Figure 7.2.

The adaptation protocol executes as message exchanges between data-
base nodes, and concurrent adaptations execute independently. The seg-
Using Virtual Full Replication

ments at each database node are incrementally updated with commutative add and remove operations. Concurrent adaptations to segments could introduce inconsistencies of the segment information at different nodes, since the list of nodes to be informed about a new allocation or removal may be stale. This is solved by letting the recovery source act as a proxy for the recovery target until the replica has been fully established. This means that: 1) User updates for objects being established must be propagated from the recovery source to the recovery target. 2) Changes to allocations at other database nodes in the system must be propagated by the recovery source to the recovery target. 3) Once the new replica is established, all other database nodes holding a replica of the established object must be updated with the new allocation by the recovery source, as a last step in the recovery process.

Replica removal

Using an estimate for the utility of keeping a replica was introduced in Section 6.2.7. For a topology in which the recovery source may be located several hops away, such as in a WSN, the utility estimate needs to be different than with a network that has the same resource cost for establishing a replica for all links.

For utility estimation in ViFuR-ASN, we use \( j \) as the distance in hops to an object replica at another database node. The highest communication cost required to keep the replica depends on the replication cost for the number of hops to the most distant other replica. The cost is \( RE_{\text{remote}} \times s_i \times \max(hopcount_j) \) where \( s_i \) is the size of the object to be kept. The communication cost of establishing a local replica depends on the distance in hops, \( k \), to the closest recovery source, and the cost is expressed as \( s_i \times \min(hopcount_k) \). The utility, \( U \), of keeping the replica is the ratio between those communication costs, as formulated in Equation 7.2. The database nodes for other replicas of the object are known from the local segments, and the hop counts for those nodes are known by the communication layer; thus \( \max(hopcount_j) \) and \( \min(hopcount_k) \) are known.

\[
U = \frac{RE_{\text{remote}} \times \max(hopcount_j)}{\min(hopcount_k)}
\]  

(7.2)
With a utility, $U$, below 1 it is beneficial to keep the replica at the time of the start of the NA-episode. As an initial estimate of the NA-episode length we use the measured A-episode length.

### 7.3 Evaluation of ViFuR-ASN

In previous work we evaluate ViFuR-A by simulation for a LAN setting (Mathiason et al. 2007). The simulation is extended in this chapter to model the ViFuR-ASN scheme.

#### 7.3.1 Simulation model

Resource usage for the ViFuR-ASN scheme in typical WSN scenario settings is examined by measuring the usage of network and storage, as well as the delays of the transactions waiting for the establishment of local replicas. We model a large surveillance and tracking scenario, as a field with hundreds of database nodes and several thousand sensors. Database nodes are randomly placed within the field at a certain average density, with close database nodes connecting to each other. Further, a minimum spanning tree of connections is added to the database nodes connections, to avoid partitioning of the network. Sensor nodes are simulated as database objects that have random locations in a certain vicinity of each database node. Update transactions are periodically executed at database nodes with primary replicas, to simulate sensor value updates to the sensor objects. Clients move over the simulated field and connect to one database node at a time, by executing transactions for sensor objects. We measure the scalability of the scheme by increasing key scale factors, such as number of nodes and clients, while measuring the resource usage. In addition, the behavior with increasing loads is also measured, by increasing the number of objects used per moving client as well as the speed of moving clients, while measuring storage, bandwidth usage, and transaction delays.

The following simulations are conducted: S1: Clients move at a certain speed across the simulated field while they connect to database nodes and execute transactions for a fixed set of sensor objects. Sensor replicas
are established from other (typically close) database nodes which have the replicas required through transactions. The number of clients is increased, while bandwidth, storage and delays are measured. In addition, the number of database nodes is increased, while bandwidth, storage and delays are measured. As a baseline comparison, the same client behavior with two alternative replication strategies is used, a fully replicated database and one with no replication of database objects. With no replication, transactions must be transferred and executed at the database node of the primary replica, and then the result is returned to the database node of the client. Simulation S1-1 uses a network with 50 database nodes, where we let 2 to 50 clients move across the field. Simulation S1-2 uses 10 moving clients, and the number of nodes is increased in steps from 10 to 100. In S2, clients move at a certain speed across the simulated field while connecting to database nodes and executing transactions for a fixed set of sensor objects. Further, clients search and request a bounded number of sensor data objects of interest in front of the path of the movement, and establish replicas for these objects at the current database node. The lookup and establishment of frontal objects simulates the tracking and following of a target in the field, which a typical behavior for the intended applications. While clients move, sensor replicas are established at the local database node, for both the fixed set and the frontal database objects. This occurs at increasingly higher request rates, where client speeds is increased and the set of objects accessed is enlarged. With increased adaptation rates, resource usage increases, and the adaptation cost of resource usage is measured as storage, bandwidth and delays.

7.3.2 Simulation settings

We model the WSN scenario according to the parameters in Table 7.3 as default settings, and vary a single or a few parameters for the simulation S1 and S2. Systems of up to 300 database nodes and 50 moving clients are chosen, since these are considered to be typical settings in a large scale WSN application. The radio range model for the database nodes is simplified, which can be justified by the fact that we simulate operation in a large open field. A few sensor objects are used and updated at each database node, and clients use a few database objects. For simulation S1 we take
10 samples for each measured value for reasonable simulation times, while for simulation S2 we take 30 samples. Using 10 samples for S1, the full replication simulation utilizing 100 nodes is computationally expensive due to the large number of update replication messages to be simulated. Each configuration of parameters for the simulation of full replication executes for several days. We choose to use 10 samples instead of 30 samples, since we observe that the confidence interval is already small enough with the use of 10 samples. Simulations for ViFuR and no replication contain a small share of the operations to simulate compared to full replication. These simulations therefore take only hours to execute. The figures for S2 show the 90% confidence interval of our measurements. The field is a square with the size $\sqrt{N\rho}$ meters, where $N$ is the number of database nodes and $\rho$ is the database node density. Using our standard parameters, with 50 nodes and one node per 20000 square meters on average, the field side will be 1000 meters.

### 7.3.3 S1: Comparing to alternative strategies

**S1-1: Number of clients**

We measure the total storage use, bandwidth over all nodes, and the summed transaction delay time over all nodes during the simulated time. Our measurements show that full replication (‘FullRepl’) requires considerably more storage and uses largely more bandwidth than the other schemes (Figure 7.3.1 and 7.3.2), as all data objects are stored and need to be updated at all the nodes. There are, however, no delays for accessing replicas. With no replication (‘NoRepl’), storage and bandwidth usage is low (Figure 7.3.1 and 7.3.2), while transaction delays are high (Figure 7.3.3). ViFuR (‘ViFuR’) uses somewhat more storage than NoRepl (used for the segment data) (Figure 7.3.1), but less bandwidth (Figure 7.3.2). ViFuR bandwidth usage is due to replication between replicas and the establishments of replicas, while NoRepl transfers complete transactions with objects at other nodes, and the transfer of transactions typically uses several hops. With typical parameters as used in this simulation, the bandwidth usage of NoRepl is high. Compared to a homogeneous network with single hops (Figure 6.8.2), NoRepl in
Using Virtual Full Replication

<table>
<thead>
<tr>
<th>Standard Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>#Database nodes</td>
<td>50</td>
</tr>
<tr>
<td>Database size</td>
<td>50 objs/node</td>
</tr>
<tr>
<td>Data object size</td>
<td>Uniform (1-128) bytes</td>
</tr>
<tr>
<td>Network delay</td>
<td>0.05-1.2 ms</td>
</tr>
<tr>
<td>#Clients</td>
<td>10</td>
</tr>
<tr>
<td>Client speed</td>
<td>30 m/sec</td>
</tr>
<tr>
<td>Simulated time</td>
<td>120 sec</td>
</tr>
<tr>
<td>Database/Database node radio range</td>
<td>100 m</td>
</tr>
<tr>
<td>Database/Sensor node radio range</td>
<td>50 m</td>
</tr>
<tr>
<td>Sensor sensing range</td>
<td>10 m</td>
</tr>
<tr>
<td>Database node density</td>
<td>1/20000 m$^2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workload</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensor update-transaction period</td>
<td>50 ms</td>
</tr>
<tr>
<td>Sensor update-transaction size</td>
<td>50 objs/node</td>
</tr>
<tr>
<td>Client transaction period</td>
<td>50 ms (Poisson arrival)</td>
</tr>
<tr>
<td>Client transaction size</td>
<td>10 objs/client</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulation S1 Parameters</th>
<th>10 samples/measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>#Clients</td>
<td>2,5,10,20,250</td>
</tr>
<tr>
<td>#Database nodes</td>
<td>10,20,30,50,100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Simulation S2 Parameters</th>
<th>30 samples/measurement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frontal sensor search angle</td>
<td>+/- 30 degrees</td>
</tr>
<tr>
<td>Frontal sensor search distance</td>
<td>200 m</td>
</tr>
<tr>
<td>Max #front sensor objects</td>
<td>10</td>
</tr>
<tr>
<td>#Database nodes</td>
<td>20, 50, 100, 150, 200, 300</td>
</tr>
<tr>
<td>#Clients</td>
<td>2.5, 10, 20, 50</td>
</tr>
<tr>
<td>Client speed</td>
<td>2, 10, 20, 30, 50 m/s</td>
</tr>
<tr>
<td>Client transaction size</td>
<td>2, 10, 20, 30, 50 objs</td>
</tr>
</tbody>
</table>

| Table 7.3: Simulation parameters     |

A WSN uses much more bandwidth, due to the higher transfer costs related to the topology and the number of hops and nodes involved in a transaction transfer. NoRepl results in very long delays for transactions missing replicas (Figure 7.3.3), while ViFuR has considerably shorter delays. The storage used with ViFuR increases linearly with the number of clients (Figure 7.3.1). In a typical workload, ViFuR-ASN reduces storage needs close to amount used by NoRepl, while bandwidth usage is a magnitude lower compared to NoRepl and several magnitudes lower than with FullRepl. Replica access delays for ViFuR using the establishments of local replicas are several magnitudes lower, compared to the access delays of using distributed transactions with NoRepl.
7.3 Evaluation of ViFuR-ASN

For each database node added, another 50 objects are added to the database, while the 10 clients continue to use the same 10 sensor object replicas. With FullRepl, the entire database is stored at every node, increasing the database at every database node with 50 objects for each such node added. For NoRepl and ViFuR, each database node will store only the 50 primary replicas. With ViFuR, another 10 replicas are stored for each client at the current database node of the client, which explains the constant difference in stor-
Figur 7.4: S1-2: Storage, Bandwidth and Delay vs. #database nodes

age between ViFuR and NoRepl with more nodes added. FullRepl sends updates for the 10 primary replicas to all other nodes, while updates are sent to one other replica with ViFuR for the load used. The network usage for NoRepl consists of moving transactions to the node of execution, and NoRepl uses up to a magnitude more bandwidth than local execution with ViFuR, including the bandwidth cost both for updates and adaptations. Transaction establishment delays for ViFuR increase with the number of nodes, and depends on the fact that a system with more nodes will use more long distance establishments, with more hops between the recovery source and the recovery target nodes.
7.3 Evaluation of ViFuR-ASN

7.3.4 S2: Scalability and adaptation

S2-1: Number of nodes and clients

For simulation S2 we measure the resource usage and delay per node. Both storage and bandwidth usage increase linearly with the number of clients (Figures 7.5.2 and 7.6.2), since the number of replicas increases linearly with the number of clients. For a database with 50 objects (with an average size 64 bytes) we expect to store 3200 bytes of data at each node. With more clients that request replicas this will be higher. The extreme case of 50 clients at 20 nodes is unlikely in a typical system, while the right side of Figure 7.5.1 is more relevant to the typical storage for 2 to 50 clients. Bandwidth usage increases slightly with the number of nodes, since update replication and replica setup use more hops (Figure 7.6.1). Delays increase linearly with the number of nodes for the same reason: there are more hops in a larger network, and increasing the number of clients does not delay transactions more (Figure 7.7.1).

Our measurements show that storage per database node increases linearly with the number of clients (Figure 7.5.2), as would be expected. Each client added to the scenario will add 10 replicas for each database objects used. The addition of nodes to the system will add to the total database size,
since each database node adds 50 unique database objects to the database, with an average object size of 64 bytes. With more clients, there will be more replicas of each object. When both the number of nodes and clients
7.3 Evaluation of ViFuR-ASN

increases, there is a considerable rise in storage. This approaches the exponential growth of storage for when using full replication, in where all objects are allocated to all nodes (Section 3.1.1). The bandwidth usage per database node only increases slightly with more database nodes (Figure 7.6.1), since there are no more replicas to be updated for more nodes. Database objects not used by clients will need no update messages to be sent from the database node of the primary replicas. Increasing the number of clients will increase the bandwidth usage linearly, as more replicas are required at the nodes of the clients. The average transaction delays per database node is close to constant (Figure 7.7.2). Possible resource limitations that could delay the establishment of replicas are not reached for the scale factors we have chosen.

S2-2: Client speed and number of objects/client

Our measurements show that storage increases linearly both with the speed that the client moves over the field (Figure 7.8.1), and when clients use larger sets of objects (Figure 7.8.2). The overall database size does not increase, since the number of nodes is constant, but the number of replicas will increase since there are more objects used by each client. Also, with
faster moving clients, deallocation will take longer to detect that a replica can be removed, which causes more replicas to be concurrently allocated in the database. The ViFuR schemes of this thesis use only the transactions to specify the current need of replicas at each node. The data-centric approach of declaring data needs by transactions creates isolation and independence.
between nodes, and keeps concurrency for establishment protocols at a relatively low-complex level. With a different approach that breaks this principle, we may extend communication for sinks that move between nodes, and use extra communication to optimize deallocation. Such an optimization could be sending the sink identifier to all neighboring nodes (within a specified radius or number of hops), at the appearance of a new sink at a node. Nodes that receive such a sink identifier message from another node about a sink that recently appeared at a node, can conclude that the sink has moved, and the replicas used by the sink at the former node may be removed immediately. Such an approach will work as an extension to ViFuR-ASN, but not for ViFuR-A, where accesses are not related in terms of moving agents, but are independent between nodes. We intend to study the refinements of protocols for WSN usage of ViFuR-ASN in future work. The bandwidth usage correlates with the number of replicas currently in the system, thus increasing close to linearly to the usage of larger sets of objects and to the speed with which the client moves (Figure 7.9.2). The delays for establishing replicas is close to constant for increasing client speeds (Figure 7.10.1), meaning that data establishment is always faster than the speeds used by the clients in this simulation. In addition, resources are sufficient to move data along the client movement. With more data to move in terms of the number of objects used by clients, the delays increase, since there are more objects to establish and remove at the nodes.

7.4 Conclusions

In this chapter, we apply constraints to the usage of ViFuR in a large-scale wireless sensor network (WSN) that has mobile clients. In this application, nodes have limited resources. Furthermore, communication links are heterogeneous, since a connection may need to use multiple hops. This type of execution environment presents a cost for adaptations that is higher compared to a LAN network, since communication links often use multiple hops of communication. Adaptation often requires the establishment of replicas from nodes that are several hops away. Also, communication links have limited bandwidth. Using a database with ViFuR as a whiteboard in a WSN
Using Virtual Full Replication

allows a tiered application that in itself enables scalability, in addition to the scalability of resource usage provided by ViFuR. With a limited search for objects in a small set of neighboring nodes, we can eliminate the need for a directory entirely, at a low search cost, as well as simplify establishment and removal protocols.

Compared to full replication, even with a limited number of database nodes as a whiteboard, ViFuR improves resource usage dramatically, such that resource usage scales for an increasing number of nodes and clients. Scalability is linear to the number of replicas required. Further, access delay for ViFuR is a magnitude lower than with no replication.

7.4.1 Results, contributions and subproblems met

With adaptive ViFuR used in a WSN with mobile clients, we see that scalability is maintained, in terms of bandwidth usage, storage, and the delay caused by adaptations of replica allocations.

We present a novel tiered approach for WSN organization, by introducing a distributed real-time database used as a whiteboard, which simplifies communication and exchange of data in the WSN. WSN nodes need no knowledge of other nodes for explicit addressing during communication, but publish sensor readings and also aggregated data in the database. Our two-tiered approach enables scalability and limits the search of objects to the upper tier of database nodes. Further, a tiered database approach allows structured storage and fault tolerance, while plain connection-based communication need to provide such distributed storage explicitly at the limited WSN nodes.

The segmentation approach of ViFuR also supports the aggregation of data, by separating raw data and aggregated data on several levels into separate segments that may have different allocations to nodes. With database nodes close to data sources, less raw data has to be replicated, effectively reducing communication needs in the WSN.

By applying ViFuR in a WSN scenario we solve subproblems 2e and 2d (these subproblems are also partly fulfilled in Chapter 6).
7.4 Conclusions

7.4.2 Discussion

The topology in use by the WSN network influences the communication cost. We choose to evaluate the usage in a WSN with a relatively sparse network, in which there are a few connections to neighboring nodes. With more connections, the communication cost becomes lower, since more connections between any two nodes use fewer hops. In a WSN, transmissions are costly operations in terms of the node’s energy. The hop count multiplies the number of transmissions for a connection, and also increases the message delays. Further, in this evaluation we do not consider partitioned networks, but assume there is always a connection between two nodes, ensured by the underlying communication layer.

The cost of the search for objects may be significant, if the hop count between two subsequently connected database nodes used by a mobile client is large. Such costly adaptations are considered in our model of the network for the node density modeled. With a greater variation in density over the monitored area, the cost is expected to vary more.
Chapter 8

Related Work

*Never do things others can do and will do if there are things others cannot do or will not do.*

-Amelia Earhart

The thesis relates to work in several research areas. The first is the area of distributed and real-time databases. There are many approaches for both non-distributed and distributed real-time databases that focus on response time, scheduling, as well as timeliness of data and data validity. There are also many approaches for distributed real-time databases that ensure timeliness and data validity under a certain known workload. Our work focuses on local execution timeliness of transactions, and on scalability and resource usage in a distributed database with eventual consistency. Scalability is achieved by considering the actual need for data. Our approach allows loads that are not fully specified, except for some basic design choices for a generic maximum load (such as the expected number of concurrent accesses to the same data object) that are matched with a few general capacities selected for the infrastructure (such as network bandwidth and storage available at each node). Within these general design choices, replication adapts to the actual load. Eventual consistency is a weak consistency model that in itself supports scalability, since only a single node needs to be accessed for updates to become valid. However, scalability is the focus of this thesis,
not the support for eventual consistency. There are alternative approaches for weak consistency in large distributed databases, where the amount of inconsistency is typically used to measure when the reconciliation of the processing of replicas needs to take place. Such metrics are, however, not needed with eventual consistency.

The second area the thesis relates to is about approaches for the replication of updates in distributed systems, particularly, approaches that allow multiple consistency models in the same system architecture, and carry out replication to a subset of the nodes, typically as a partially replicated database. Distributed databases, which support a multiple data consistency model, are to our knowledge rare in the literature. A probable explanation is the common usage of distributed transactions, rather than having a data-centric approach. Distributed transactions suffer from the complexity of process shipping, where transactions execute (partially) at remote nodes, which includes the problems of resource allocation and synchronization for remote transactions. With a data shipping approach, the update synchronization and data locking complexity of transactions are kept at the local node. Generally, data replica allocation suffers from the file allocation problem (Casey 1972, Chandy & Hewes 1976), that is, how to allocate a partially replicated database such that the allocation of replicas is optimized for nodes that have the lowest cost of communication and data access. Approaches involve tradeoffs between communication cost and the cost of the number of replicas. Often, full consistency between a few key replicas is used, requiring a distributed agreement protocol for updates to any of the replica. Such approaches have high performance when there are a small number of replicas involved in updates and a high number of replicas when data is read (Helal et al. 1996). The number and allocation of replicas are decided based on the characteristics of the application, considering the amount of readers and writers and their location in the network. Remote operations over communication links are typically used for a majority of accesses to data, while our approach only needs local data for timeliness of access, and uses weak consistency.
8.1 Scalability and data storage

The third research area that this thesis relates to is the area of (distributed) caching and database buffering, as well as virtual memory, and garbage collection. Our adaptive allocation of data allocation is closely related to the local caching of data objects to be used at the local node. An important distinct difference in our work is that local availability is an absolute requirement for local timeliness, and that delays in setting up replicas from remote nodes occasionally occurs. The typical real-time application that we aim for uses the same set of data for some extended period (as in a certain mode of operation), before changing to the use of another data set. This resembles a database buffering problem more than an instruction or data cache. A cache miss for a locally cached data object is a typical execution model for generic caching systems. Caching is a performance improvement feature for the access of data, not an absolute requirement.

Finally, the thesis relates to the application area of Wireless Sensor Networks (WSN), which we use in our work. Our approach relies on existing work of searching for data in a WSN, and gives a novel data-centric approach for multi-tiered data propagation.

8.1 Scalability and data storage

Scalability as a concept is used in many areas of research, but the availability of a generic definition and a theoretical framework that uses metrics is limited. However, scalability concepts are well developed in a few research areas, such as in parallel computing, for shared virtual memory, in the design of distributed architectures, and for network resource management. We present some of the generic approaches for the management of scalability as a concept in Section 2.5.

Satyanarayanan (1992) evaluates the distributed file systems Andrew and Coda, and draws conclusions regarding a set of design principles that are essential for scalability in terms of nodes and users in such systems. According to Satyanarayanan (1992), there should be a small nucleus in the system that changes slowly. Computation should be done close to the client, and files should to be cached whenever possible for high availability. Caching is the feature that contributes most to scalability in distributed file systems.
In addition, there should be a clear separation of clients and servers, as well as a functional specialization, and minimization of the system-wide knowledge. A scalable file system should also use properties that are known about the data. Only a few nodes must be trusted, to ensure security. Furthermore, operations should be grouped into batch processing when possible, particularly for large cached entities. Finally, updates should be distributed as differences between old and new values. Our work is related to these principles in that it follows several of the guidelines: usage of local caching for scalability, minimization of global knowledge, establishments involving a recovering client that is supported by a server with an existing replica, and the use of batch communication when possible. Our work differs in that we provide real-time properties for timeliness of transactions.

8.1.1 Scalable data stores

Many scalable data stores in literature share the properties of using local operations, or fully distributed algorithms for data management. Nodes are highly independent and contribute resources, as well as providing scalable search for data. Peer-to-peer systems are typically highly scalable, and it is common to use distributed hash tables to enable scalable search for data, as used with OceanStore (Kubiatowicz et al. 2000), CAN (Ratnasamy et al. 2001), Pastry (Rowstron & Druschel 2001), and Chord (Stoica et al. 2003). ViFuR instead uses a distributed name service that can be quickly accessed at a bounded set of nodes. This is efficient when the allocation change rate is reasonable. The removal of replicas requires that a certain minimum number of replicas are ensured, which needs a coordinator to allow removals, as implemented by our object directory. In peer-to-peer systems, all replicas are allowed to disappear, while ViFuR ensures availability by removal coordination of the directory.

Recent large-scale data stores include BigTable (Chang, Dean, Ghemawat, Hsieh, Wallach, Burrows, Chandra, Fikes & Gruber 2008) that is used by several different types of internet-scale Google applications, which all have different requirements on a large data storage. BigTable is designed to be highly scalable and flexible, for usage with any structured data, and to reduce latency. It uses several different scalability enablers: hierarchical
8.1 Scalability and data storage

architecture with back-end server processing, local allocation knowledge (a "library"), a single master server with a single master replica, and dynamic servers ("tablets") that are dynamically added or removed from the system. Further, it uses Bloom filters to find data, and local caching for improved read performance, and it sets up tablets by using recovery processing. Our database-based approach uses many similar features, although not primarily intended for internet-scale usage. Instead, our approach focuses on the local real-time execution of transactions, and full consistency at the local node.

8.1.2 Scalable replication

There are several file replication architectures available in literature that replicate updates with relaxed consistency. This in itself is a scalability enabler, since coordination and agreements on updates are less complex. In the Gossip architecture (Ladin, Liskov, Shrima & Ghemawat 1992), close neighbor nodes periodically exchange updates, and where the message order is preserved at every node for a correct update order of the data. There is no notion of storing subgroups of files with an individual degree of replication.

The Bayou (Terry, Theimer, Petersen, Demers, Spreitzer & Hauser 1995) and Coda (Satyanarayanan, Kistler, Kumar, Okasaki, Siegel & Steere 1990) systems provide scalable file replication by using locally cached copies of files in networks, to allow updates to highly-available local client files, also during partitioning. Their primary design goal is to support independent updates and partitioned operation. Bayou exchanges updates pair wise between nodes, and eventually all updates are propagated to all client nodes. Using Bayou is relatively complex, since the user needs to be aware of the state of updates. Updates can be tentative or committed, so that users could use preliminary updates that have no guarantees of being consistent with all other replicas. Also, the user must deal with dependencies of updates, and provide conflict resolution procedures. The Coda system uses a scalable architecture that separates clients and servers, and manages dynamic memberships. The clients autonomously use locally cached copies, and each local replica at a partitioned node is optimistically updated. Conflicts are found and resolved at reconnection to a server. Both Bayou and Coda scale due to the use of cached replicas for high availability. They also need to
manage cache coherence, just as with ViFuR. However, they are not distributed real-time databases with timely transactions, and there are no subgroups of the files.

With epidemic replication, each node re-propagates updates while preserving the time order of updates, until all the updates have reached all the nodes. It ensures one-copy serialization of updates at each node by establishing the order between all pairs of updates, for example, with the use of version vectors (Popek, Walker, Chow, Edwards, Kline, Rudisin & Thiel 1981). Epidemic replication is suitable for database replication (Agrawal, Abbadi & Steinke 1997), where the log at each node can represent the order of a pair of ordered updates. Nodes exchange logs locally between neighboring nodes, which ensures scalability in terms of the number of nodes. There is no conflict resolution mechanism, and no selective replication of updates for groups of data objects. The overhead of version comparison grows linearly with the number of data items, which limits scalability in a system where all data needs to be available at all nodes. Rabinovich, Gehani & Kononov (1996) propose an epidemic replication protocol that avoids much of the copying of updates and scales with the number of data items actually copied during update propagation, but that still has all data items allocated at all nodes. Holliday, Steinke, Agrawal & Abbadi (2003) present epidemic partial database replication for unreliable networks that allow replicas of data objects to a few nodes. Transactions may read and update replicas at remote nodes, while replication adapts by re-caching local replicas of data. This approach has many similarities to our approach. However, it allows remote transactions in unreliable networks, which may cause unbounded execution time for transactions in a best effort network. Also, there is no knowledge of how to find the replicas needed, and thus adaptation time is unbounded.

The ADR replication algorithm (Wolfson & Milo 1991, Wolfson, Jajodia & Huang 1997) provides adaptive allocation of objects in a distributed database, to match the replication schema to read-write patterns of database accesses. It adapts the replication schema to maximize performance and to minimize the cost of communication, and it can be used for both strong and weak consistency replication. Replicas are not necessarily located at each
8.1 Scalability and data storage

node of access, but can be located at other close-by nodes, which involves communication in the update processing. The scheme allows remote transactions in optimization for low resource usage of allocations, without considering timeliness of transactions or consistency between replicas, as with ViFuR. With ADR, remote transactions may have unbounded update delays in a best effort network.

8.1.3 Group communication

In this thesis, we separate the issues of segment membership and group communication. Group communication and ViFuR are orthogonal, since group communication is a scalability enabler that is independent of ViFuR, and ViFuR is a scalability enabler that is independent of group communication. Even if they are orthogonal, multi-cast communication, group communication and group membership management are highly related to our work, since update replication for segments may benefit from group communication. We assume that the underlying communication layer provides communication with other nodes, either as point-to-point or group communication message primitives, and in our evaluation we assume point-to-point communication. Multi-cast primitives are available in many networks (e.g. IP or Ethernet networks). They are weak cases of group membership services, since there is no support for the dynamic adjustments of groups.

A full group membership service includes four main tasks (Coulouris & Dollimore 1988): An interface for changes of group membership; implementation of a failure detector; notification of group membership changes to members; group address expansion. Group membership is intended for process membership in distributed systems, while our approach is data-centric. By using ViFuR on top of a communication layer that provides group communication primitives, the overall architecture approaches a full group membership service: There is an interface for communication within the group; This includes dynamic groups that are changed over time, and changes that are allowed during ongoing communication in the group, such that all members are informed of the new group layout, and updating an object in a segment that expands the address into the addresses of all other replicas of the object. We have not explicitly dealt with member failure
Related Work

detection, but by using acknowledged change messages for segmentation changes (as described in Section 6.2.4), node failures can be detected, and the setup approaches a full group membership service.

There are several approaches for group membership services that have similarities with ViFuR. The Ficus replicated file system (Guy, Heidemann, Mak, Page Jr., Popek & Rothmeier 1990) uses partial replication and weak consistency. This system presents several of the challenges later pursued by others, regarding deferred replication, update conflicts, and scalability through partial replication. The approach uses group communication primitives, while ViFuR is separated from the communication layer, using communication layer interface primitives.

Approaches for group membership services to control selective replication are used for scalability improvements in several distributed database approaches, which typically use distributed transactions and support strong consistency only (Jia, Kaiser & Nett 1996, Alonso 1997). We regard group communication as orthogonal to our work, and also an opportunity for further scalability improvements when applying ViFuR in future work.

8.2 Scalable replicated databases

It is essential to have localized operations in order to achieve scalability in replicated systems. With localized operations, processing at a node relies on the local node, or a few, preferably highly available neighboring nodes. Secondly, for scalability, nodes should contribute with resources at the same rate at which resources are being used when the system is scaled up. For replicated databases that must use strong consistency, scalability is very difficult, since all the replicas need to agree on any updates. Approaches such as quorum consensus (Gifford 1979) allow updates when a majority of the nodes are available for a distributed agreement. For large-scale systems it can be hard to reliably connect to a majority, since the size of the majority group increases at the same rate as the number of nodes.

For scalability, it is necessary that the resources (such as the number of nodes involved), used in an update, grow slower than the scale factor at which the system is increased. For distributed databases using eventu-
al consistency and detached replication, only a single replica need to be available for an update to be valid, which is far more scalable than with a majority. Further, even with a bound on a single replica, resources used in update replication must be bounded. With a bound on the set of replicas to receive updates, the system scales with size of that set. For weak consistency schemes, there must also be bounds for resource usage in conflict detection and resolution for conflicting updates.

8.2.1 Replication and consistency

Partial replication is commonly used in distributed databases, but with many approaches, the data user needs to know where data can be accessed. Early work focused on reducing the manual design effort of data allocation. Mukkamala, Bruell & Shultz (1988) present an approach for partial replication and allocation of data based on known accesses. Data is grouped to meet the Data Application Problem (c.f. The File Allocation Problem) (Chandy & Hewes 1976), for high availability and load balancing by minimizing the communication load through the use of heuristics. Remote accesses are allowed but real-time properties are not considered, except from the minimization of the access waiting time by reducing the communication cost.

There are several approaches in literature that apply consistency control for scalability and reduced resource usage with replication to few nodes. Some approaches allow multiple consistency models to coexist. Yu and Vahdat (Yu & Vahdat 2002) present an approach to minimize replication cost by replica placement on large networks, such as the internet. ‘Consistency units’ are units for replication that may use separate consistency models for each unit, so that strong and weak consistency can coexist in the same replication framework. The replication of units originates from the actual data need. The approach supports data availability and improves efficiency at the cost of a specified lowest level of consistency, but provide no real-time properties, nor guarantees of local data availability as with ViFuR.

There are several approaches for replication models used in large scale distributed databases. In contrast to our peer node approach, the hierarchical asynchronous replication protocol (HARP) (Adly, Nagi & Bacon 1993)
orders nodes in a logical hierarchy of node groups such that each node communicates with a few other nodes, and replicates to neighbor nodes with two different replication methods: one that uses global commit and another method with weak consistency that propagates updates independently and where updates eventually reach all nodes. Nodes are organized in a hierarchy such that updates are propagated along a dissemination tree, which enables ordered propagation of updates. The hierarchy can be dynamically changed and any replica can receive updates, but there is no explicit focus on real-time properties as with ViFuR.

Sivasankaran, Ramamritham, Stankovic & Towsley (1995) use characteristics of data accesses in a centralized database for data placement in buffer memory or on disk, and for consistency control. Two static consistency models coexist, such that a critical and a non-critical data set are distinguished in the database. Properties of data are used for the data management. In contrast, ViFuR allows multiple coexisting consistency models in different segments of a distributed database, and objects can be reassigned to other segments and other nodes when data needs change.

8.2.2 Distributed real-time databases

Many distributed databases enable mobile nodes that frequently connect and disconnect to and from the network. The Ward replication model (Ratner, Popek & Reiher 1996, Ratner 1998) presents many components needed by a scalable distributed real-time database. Scalable version vectors (dynamic version vectors) are used for consistency management in optimistic replication, and to recover from network partitioning. Ward controls replication to a subset of the nodes without using any coordinator. It enables scalability in many ways, but gives no timeliness guarantees, globally or locally.

For distributed databases where data validity in temporal real-time distributed databases is central, the ORDER scheme (Wei et al. 2004) ensures data deadlines by coordinating replication updates between groups of nodes (‘cliques’) where coordinator nodes are periodically updated by sensors, and then updates are replicated to users of sensor data in other cliques. Update periods for groups of nodes using the same data are coordinated to consider the differentiation in link costs, such that slow links are used less frequently.
There is no support for adaptive allocation of data or independent updates. There is also no consistency control schema to allow independent updates, but this is typically not needed in a sensor network application with one writer only.

8.3 Allocation and availability

There are many approaches in the literature that handle the tradeoff between high availability for efficient execution and the cost of high availability, also including adaptive allocation of replicas over time. Typically, such approaches minimize the communication cost and provide no real-time properties.

8.3.1 Adaptive replica placement and caching

The need for adaptive change of allocation and replication of database objects in (large) distributed databases has been identified in literature (Wolfson et al. 1997), including relational (Brunstrom, Leutenegger & Simha 1995) and object database approaches (Lin & Veeravalli 2003). There is a tradeoff between communication cost and availability, and real-time guarantees are not considered. The adaptive allocation of data that considers performance can be found in the areas of cache coherency and replica allocation (Alonso, Barbara & García-Molina 1990) (Wolfson & Huang 1998) (Park, Lee, Lim & Yu 2001), database buffer management (Huang & Stankovic 1990) (Jauhari, Carey & Livny 1990) (Datta, Mukherjee & Viguier 1998), and also virtual memory (Denning 1970). Our approach for the adaptive allocation of data objects (both allocation and deallocation) relates to these areas, but ViFuR always requires a local replica for transaction execution, and transactions always execute locally. With ViFuR, there is no complete backup storage, since other replicas are spread over the nodes. ViFuR prioritizes the availability of local replicas for current data needs, but also more aggressively deallocates replicas due to the high cost of resource usage in keeping them.
The Cache Only Memory Architecture (COMA) is a large-scale shared-memory cache approach used for multiprocessor systems (Stenström, Joe & Gupta 1992). COMA replicates cached data at the main-memory level of the processors. It uses a hierarchical directory structure to locate data at cache miss, and must prevent removal of the last cached replica. This is similar to ViFuR and both schemes perform best when there is a cohesive set of data is being used. For instance, groups of needed data can be setup together, which shortens the average waiting time for data. They both suffer from the latency of setting up local data copies, thus infrequent changes to the local set result in shorter access times on average. COMA and ViFuR differ in that ViFuR not only stores data in shared distributed memory, but uses transactions to allow independent updates and controls replica consistency. COMA is intended as memory management unit (MMU) implementation, while ViFuR is a part of distributed database replication mechanisms.

In distributed databases, there are different tradeoffs for read and for update replicas. The same object may have both types of replicas. For read replicas, there is an advantage in having many replicas to improve availability and reduce access time, while updating a large number of replicas is costly. There is an advantage in having a small number of updated replicas, since there is a cost in coordinating updates to keep them consistent, while availability is important for short update times. For both read and update replicas, a low cost for availability is essential.

A seminal work by Plaxton, Rajaraman & Richa (1997) provides a framework and an analysis for the distributed allocation of data replicas. The approach allows replicas of used objects to be closely available, and uses replication trees for propagating updates to replicas. Allocations change over time and there is functionality that finds addresses of replicas at the cost of a small storage overhead (auxiliary memory). The approach allows local processing and updates to data replicas, such that only local or neighbor nodes are involved. The framework does use a distributed database with transactions, but is modeled as distributed shared memory. The analysis involves memory usage (“data” and “auxiliary” memory), the cost of adding and removing objects and nodes, as well as the number of changes to auxiliary memory when adding and removing nodes. However, there is
no analysis of communication costs. The work is highly theoretical, with no implementation or simulation. In addition, the analytic model has not been used with any typical application parameters. Finally, the approach does not consider any timeliness properties of adaptation.

One important challenge in peer-to-peer systems is allowing the dynamic change of the overlay network, made up of the contributing nodes. Chen, Katz & Kubiatowicz (2002) use a dissemination tree to control the distribution of web content among a few nodes on a large network (up to 5000 nodes), where an improved version of Application Level Multicast (ALM) is used for addressing at the overlay level. Standard ALM uses one central node for dissemination-group management, and there is a single source of each data object and multiple cached copies on clients. Dissemination trees are used at object level, and dissemination is done with limited knowledge about the network topology. This approach tries to minimize the number of replicas, reduce the overall resource usage, and uses a bounded number of hops from the single source copy to the cached copies. The improvement of ALM is in the use of finding data objects, where an approach similar to that of Tapestry (Zhao, Huang, Rhea, Stribling, Joseph & Kubiatowicz 2004) is applied to search for data, rather than using a central node for lookup. ALM is intended for web content but has many similarities to our work, since it dynamically tries to minimize the number of replicas. QoS is supported such that the timeliness constraints of data users are considered. However, there is no assurance of local timeliness of data access, since communication is involved in such accesses. Dissemination trees are used with a master replica, and also based on a global time, managed by the NTP-protocol. We allow independent updates and need no global time synchronization. Lin & Veeravalli (2006) propose the DWM algorithm to minimize service cost for adaptively allocating objects in a distributed system. It uses a central control unit (CCU) to serialize allocation and deallocation decisions, but the CCU becomes both a single point of failure and a bottleneck for scalability. In contrast, ViFuR-A uses commuting operations and a directory to avoid this. Our work shares the goal of bounding the number of replicas in a system, and we minimize delays caused by the network and that jeopardizes timeliness of accesses. In the evaluation of our approach, we have recognized the
problem of using a central lookup service. Thus, in our approach for ViFuRASN, we use a bounded search instead, to find the objects. In future work, we intend to pursue a replicated directory service, as proposed in this thesis, to reduce the problem of a central service. Our approach does not consider a global optimization problem, since local availability and timeliness is of more importance for distributed real-time systems than a minimum resource usage cost of communication.

Peer-to-peer file distribution schemes need high availability and functions to find files of interest in very large distributed systems. They are highly dynamic, with nodes that are frequently added and removed. In addition, dynamically placing replicas for availability improves performance. Adaptive Replication has been proposed for Peer-to-Peer Systems (Gopalakrishnan, Silaghi, Bhattacharjee & Keleher 2004), and for system where the allocations of file replicas are adjusted on demand. One goal of peer-to-peer systems is to increase availability and the number of replicas. Such networks optimize read-availability, in particular for file distribution schemes. In a distributed database, both update and read operations are combined in the same transactions. Furthermore, besides high availability a bounded response time is also a parameter that needs to be optimized on for distributed real-time systems.

The ADRW algorithm is used for on-demand adaptations of object allocations in Distributed Database Systems (Lin & Veeravalli 2003). ADRW differentiates read and write requests of random transaction access patterns and globally optimizes data object allocations over time. The scheme is aimed at fully consistent databases and is carefully analyzed for competitiveness, in terms of cost functions for accesses involved. ADRW caches objects locally from the closest node with a replica, but local availability is intended to improve performance, rather than retain local replicas for timeliness of subsequent transactions. Thus, contrary to ViFuR, replica consistency is more important than timeliness.

8.3.2 Data-centric and whiteboard style of communication

Large-scale distributed databases that have weak consistency are suggested as a useful approach in scalable and dynamic application scenarios (Helal et
al. 1996). A data-centric whiteboard approach decreases coupling between collaborating nodes, since processing is not distributed, nor is it dependent on reliable communication and connectivity. A tuple space is one abstraction for a distributed associative memory that allows publication of key-data pairs in a distributed system, as a shared memory with a single master, for availability and inter-operability (Gelernter 1985). Using a distributed database for a whiteboard style of data shipping, as used in this thesis, provides a distributed shared memory with database properties (transactions for isolation and concurrent updates), and can be regarded as an extended tuple space abstraction.

A distributed database has been suggested as a suitable infrastructure for emergency management (Tatomir & Rothkrantz 2005) that could be used as a whiteboard for publishing current information about the state of a mission, supporting situation awareness from the information assembled from all the actors. Using a database with transactions ensures consistency of the information, and also avoids the necessity of specific addressing between nodes of the communication infrastructure. Ethnographical field studies (Landgren 2005) show that such an infrastructure supports sense-making by enabling humans to interact. It also provides support for the chain of command and strengthens organizational awareness, which is essential for the success of the mission (Oomes 2004). In such an infrastructure, actors may have access to all the information when needed, but each actor will mostly only use parts of the information for their local actions and collaboration with close-by peers. We argue that a distributed database with virtual full replication is a suitable approach for such an infrastructure to allow scalable and adaptive resource usage.

### 8.4 Database buffering and caching

For hard real-time databases, predictability is more important than performance, since deadlines must not be missed. For soft real-time databases, all performance improvements that can be gained will reduce the transaction value loss caused by a missed deadline. Many real-time databases make use of local buffering of data to improve performance. In hard real-time
databases, the entire database is stored locally for predictability. For soft real-time databases, local buffering of the most frequently used data in local main-memory will improve the missed-deadline-ratio, even with a relatively small buffer, if the hot-spot is small. Further, for databases in general, main-memory buffering is a way to improve performance and avoid disk accesses. Also, local buffering in distributed databases fills the same purpose.

A number of buffering schemes exist, both for local and distributed databases. For the efficient usage of local memory (in terms of transaction performance), a range of caching and buffer replacement schemes are available in the literature, starting with relatively simple schemes such as the least-recently-used (LRU) scheme (Franklin, Carey & Livny 1992, O’Neil, O’Neil & Weikum 1993). LRU-based schemes manage past references to data in order to estimate future expected accesses. The management of references is typically part of a replacement policy, taking decisions when to deallocate data from the local buffer, allowing new data in it.

Database buffering schemes have many similarities with caching schemes in general, both for data and instruction caching, as well as schemes for virtual memory (VM) (Denning 1970). Many early database buffering approaches use principles from VM. Effelsberg & Haerder (1984) outline the differences between virtual memory paging and database buffer management, explaining that the locality of reference patterns is different, and as a consequence, reference management and replacement need to be different. Database applications typically have skew access to the database, such as with b-c accesses, which can be utilized for buffering the most frequently used data. There is, however, less locality in database accesses, compared to the access of instructions or memory in virtual memory systems. For database buffering, it is harder to meet data needs of the near future by sequentially loading data into the buffer. Effelsberg and Haerder also point out that optional locking of buffer objects can be used to keep objects in the buffer regardless of accesses counted, to guarantee availability for certain critical data, which is similar to pinned object used by ViFuR. Some systems classify data into multiple data sets, one where data is sequentially accessed, and another with random locality. Sacco & Schkolnick (1986) present a buffer management scheme for relational databases that uses a hot-set model based
on database queries to determine the optimal buffer space.

For buffer replacement in real-time databases, Carey, Jauhari & Livny (1989) introduce priorities for traditional Least Recently Used (LRU) schemes. The work shows the importance of admission control and scheduling of CPU and disks. Huang & Stankovic (1990) find that conflict resolution is a key factor for performance with hot data. They also find, for the real-time systems studied, that complex buffer management approaches are not better than simple approaches such as LRU. The PAPER algorithm by Datta et al. (Datta et al. 1998) uses priority-based buffer management and pre-fetching of data. The authors conclude that LRU approaches are appropriate when buffers are large, but also that pre-fetching gives limited improvement. The LIRS algorithm (Jiang & Zhang 2005) is a generic, simple and effective LRU-based approach that uses recency of accesses with reference counting. There are many replacement approaches available for specific access patterns based on LRU, a good survey of which can be found in Goh, Shu, Huang & Ooi (2006).

In our approach, we have chosen to use a low-complex reference management that is not optimized to any specific load characteristics, except for utilizing common hot-spot behavior of accesses. Malik et al. (2005) present the simple and effective BYHR approach, which we have used as a base for our approach 6.2. Further, we do not use implicit prediction and pre-fetch for future data needs, but let the application issue prospective transactions to shorten the waiting time for setting up local data on-demand, prior to the actual timely transaction. This can be used by the user application, if there is knowledge of future data needs.

8.5 Communication in wireless sensor networks

Research on communication in wireless sensor networks has focused much on energy-efficient communication and routing, for communication from source nodes to sink nodes at the edge of the network. Several data-centric approaches are available, but most use a data-oriented communication approach for a single sensor node level only. Our usage of ViFuR in WSNs takes a multi-tiered whiteboard approach of communication, with a dis-
tributed database layer on top of any simple communication layer that provides point-to-point connection between pairs of nodes. The multi-tiered database layer provides a way of publishing and aggregating data within the network, for high in-network availability and for scalable communication.

8.5.1 Data-centric communication in WSN

Several alternative data-centric communication approaches exist for WSNs. TinyDB (Madden et al. 2005) and the COUGAR database system (Bonnet et al. 2001) are approaches that present a database view of a WSN by mimicking a database interface to the WSN. The queries of users are declared at the network edge, and distributed for processing at sensor nodes. There is no structured storage in TinyDB, as with a regular database, and edge queries are static, since the distributed queries are compiled from user queries offline. COUGAR provides persistent views of sensor data, by applying distributed queries that return reduced time-series of data from sensors. Distributed queries filter or aggregate data streams close to the sensors, such that communication is reduced and less data is propagated over the network.

The Data-Centric Storage (DCS) by Shenker et al. (Shenker, Ratnasamy, Karp, Govindan & Estrin 2003) is a companion method to data-centric routing that stores named data and uses data names to describe data content. Content that is similar is stored at the same location or at close nodes, such that both queries and updates are propagated towards a limited number of access points. DCS uses a peer-to-peer-style for the looking up of data, utilizing a Geographical Hash Table (GHT). GHT is a distributed principle for finding named data that directs data lookup to the nodes that have matching data names, which avoids flooding of the network. With DCS, data is accessed remotely most of the time, while with ViFuR data is accessed timely at the local node. With DCS, all nodes are assumed to have the same capabilities, although it is recognized that some nodes may have more resources, while with ViFuR there are nodes at the upper tier with more resources. Multiple tiers are possible with DCS, but not pursued. Furthermore, DCS is intended for large-scale systems, but not evaluated for scalability.

DSWare (Li, Lin, Son, Stankovic & Wei 2004) is a data-centric service middleware for WSNs, using DCS and own improvements. Data is replicat-
8.5 Communication in wireless sensor networks

ed, by caching data replicas along a path in where most queries are found to be issued. Such data allocation improves performance for query issuers. For filtering and aggregation, DSWare uses an event service to detect when to send data over the network and what to send. Sensor readings are correlated within groups of sensors to improve confidence in sensor readings, and groups also manage faulty sensors. Events to be detected are predefined and static, and there is no evaluation for scalability of DSWare. ViFuR allocates replicas for local and timely access, while DSWare allocates replicas to improve performance. Such placement of replicas is always a tradeoff between number of replicas and performance (or cost of access). In a system with timely transactions, any involvement of network communication during data access jeopardizes timeliness. In addition, a wireless network is even more unpredictable than a wired network.

TTDD (Luo, Ye, Cheng, Lu & Zhang 2005) is a tiered approach that assigns a subset of nodes in a course grid within the WSN, where nodes of the grid act as brokers for setting up communication between distant sensor nodes. The grid overlay reduces flooding of searches for data in the network, since the grid nodes represent the sensors in its grid. Multiple mobile sinks are possible, and up to eight sinks were used in the evaluation presented. TTDD is self-organizing, such that grid maintenance and communication paths are dynamic. TTDD sets up communication per source-sink connection which can be expected not to scale well. TTDD is typically used in a homogenous network of sensor nodes. ViFuR, in contrast, has more powerful nodes at the upper tier. An upper tier of ViFuR can be regarded as a course-grained overlay grid, but the organization of ViFuR releases the strong connection between data publishers and data users, while TTDD maintains the link between each pair of communicating nodes.

The Tenet tiered architecture (Gnawali, Jang, Paek, Vieira, Govindan, Greenstein, Joki, Estrin & Kohler 2006) uses more powerful “Master” nodes to aggregate and process parts of the WSN application. Tenet distributes small tasks (“tasklets”) over the nodes in a WSN, such that more demanding tasks are executed at more powerful nodes, and powerful nodes delegate simpler data collection tasks to limited sensor nodes. Scalability is supported since operations are kept local. Tasklets are available for usage through
a tasklet library. There is no structured storage or distributed database in the architecture, as with ViFuR, and real-time properties are not addressed.

Koubaa & Alves (2005) present a work-in-progress that proposes a two-tiered approach for communication in WSNs. The approach ensures timeliness at the sensor level, under a max radius (a bounded number of hops). Data is implicitly stored, but there is no distributed database used as with ViFuR. It is stated that the approach presented has a feasible implementation complexity, although no actual implementation seems to be done.

8.5.2 Searching for data in WSN

Several approaches to find data objects are available for WSNs, many of them inspired by search in peer-to-peer networks. A search can be done by naively flooding the network and waiting for a reply, but this does not scale well. Many approaches limit the search space by using a specification of the required data or data properties to find data in a large network, rather than using an object identifier to look up data. Such approaches include a search by a limited flooding of the network (Ye et al. 2002), Directed Diffusion (Intanagonwiwat, Govindan, Estrin, Heidemann & Silva 2003) and Geographic Routing (Ratnasamy, Karp, Shenker, Estrin, Govindan, Yin & Yu 2003). These protocols use the fact that sensor value objects are less useful when they are not tagged with a location and possibly additional sensor and data properties. A search for data thus makes use of such information to improve a flooding search. ViFuR has an approach for a local search, but also applies directional search to use data, for example, for areas of particular interest. We intend to improve the currently applied directional search by using a DHT-based approach in the future.

In peer-to-peer networks, the number of nodes is potentially very large, and a flooding based search may become very expensive. Instead of searching through an unknown number of nodes, many of these networks use a lookup service. The Chord lookup protocol (Stoica et al. 2003) is commonly referred to in the literature, since it is seminal work in the area of distributed lookup protocols using a variant of consistent hashing to lookup objects in very large networks. The protocol generates a key for a com-
8.5 Communication in wireless sensor networks

bination of file and node. Several other similar protocols exist, such as CAN (Ratnasamy et al. 2001), Pastry (Rowstron & Druschel 2001) and Tapestry (Zhao et al. 2004). Using Chord, each node needs to maintain routing information about \( O(\log N) \) nodes (where \( N \) is the number of nodes) in a non changing state, and communication with \( O(\log N) \) nodes is needed to resolve the hash key for a file on a node. Chord cannot handle partitioning, since it organizes local information such that each node will be a link between a few other nodes. Consistent hashing tends to balance the load well, and also allows inexpensive changes since only a fraction of the keys are changed when nodes join and leave. Chord has been extended to approach query latency and to improve load balancing: LAR (Gopalakrishnan et al. 2004) sets up and removes files on demand, but there is no guarantee of local availability.

A lookup protocol based on consistent hashing, to look up of files in peer-to-peer networks, is a scalability approach, since all operations are localized to a few nodes. ViFuR-A uses a similar, but orthogonal, approach that applies a replicated directory available at a small set of nodes. ViFuR-A needs the manual assignment of directory nodes for the name service, since directory nodes must be highly available to establish objects. In future work, we intend to make use of consistent hashing for the lookup of objects in a distributed database.
Chapter 9

Conclusions and Future Work

However beautiful the strategy,
you should occasionally look at the results.
- Winston Churchill

In this thesis, we explore Virtual Full Replication (ViFuR) as an approach for scalability in such distributed real-time databases. ViFuR gives a perception of full replication that allows any database user to access any object of the distributed logical database at any local node of execution. Full replication and eventual consistency provide timely execution of transactions, as well as a database interface that gives database application simplicity, by simplifying data access. This allows the user to assume full access at the local node for any database object, while ViFuR provides the same database interface and also makes the database scalable and flexible, by ensuring that objects are available when needed. The simplicity of using a fully replicated system entails a high growth of resource usage as the system grows. For bandwidth usage and processing time for replication, resource usage increases exponentially with the number of nodes added, as the database is duplicated again at each new node. To enable large-scale distributed real-time databases, we use ViFuR as an approach for scalability. Full replication allows arbitrary access to any data object at any node, which a ViFuR approach with static allocation of replicas cannot provide. An adaptive app-
roach that changes allocations to meet evolving needs can provide flexibility for access of arbitrary data objects. ViFuR manages adaptive allocation of replicas, such that any new need arising during execution time will be met by new replicas established at the node where required. Further, replicas not in use are de-allocated to keep the number of replicas small to maintain scalability, while allowing flexible data accesses.

The overall result from this thesis is the three ViFuR schemes developed for a distributed real-time database with eventual consistency, as well as evaluations of these schemes which show that ViFuR enables large-scale distributed real-time databases with timely transaction execution. We present algorithms and a model architecture for ViFuR, an adaptive segmentation of the database, as well as an evaluation of the approach for adaptive changes to such segmentation. Further, we apply and evaluate ViFuR in a typical large-scale distributed scenario, a Wireless Sensor Network (WSN). This reveals that ViFuR is suitable to use as a whiteboard in typical large-scale real-time database application. We find that such a database scales with the amount of concurrently needed replicas, and that a ViFuR scheme maintains scalability and flexibility over time for typical and changing accesses.

9.1 Contributions

The following contributions are the detailed results of elaborating and exploring ViFuR.

Virtual Full Replication:

1. We elaborate Virtual Full Replication by segmentation as an approach for scalable and flexible whiteboard communication, using a distributed real-time database. Further, we formally define both ViFuR and segmentation, and show how segments are formed by using object properties. We exemplify how to use combinations of object properties for the segmentation.

2. Multiple segmentations may introduce requirements from conflicting
properties for the segmentation of the database. By introducing and using rules, such conflicting properties can be found and resolved, manually or automatically. Using consistency checking rules ensure that segments and segmentations created are consistent, and that they are also consistent with explicit requirements of the database application, and the execution environment.

3. We present requirements an an approach for a model architecture that supports a segmented database, also including the usage of multiple properties and segmentations, and which has replication mechanisms for multiple and concurrent consistency classes.

Static segmentation:

4. An efficient and scalable algorithm is presented for static segmentation of a database, based on pre-specification of accesses through transactions executed by the application, for hard and soft real-time database applications. This algorithm has $O(o \log o)$ computational complexity and $O(o + s)$ storage complexity for $o$ objects and $s$ segments for each segmentation, and can generate multiple segmentations for different purposes on combinations of properties.

5. We describe the implementation of a table-based approach for static segmentation based on pre-specification of known accesses. Further, in such a table, rules can be applied for consistent segmentation. Our analysis of scalability shows that a vast amount of bandwidth can be saved by considering the actual need for data, compared to full replication, since resource needs grow linearly with the number of replicas, rather than with the number of nodes: For bandwidth, the number of transactions often depends on the number of nodes. For $k$ replicas, bandwidth usage for an update to all objects need $O(k*n)$ messages for $n$ nodes, while full replication needs $O(n^2)$ messages for the same number of updates to the database. With $o$ objects, full replication needs to store $O(o*n)$ data objects, and ViFuR need to store $O(k*o)$ objects. For ViFuR, the processing time for the replication of updates grows with the number of replicas, $k$, and in addition, fewer replicas
also reduces the size of the potential conflict set, with reduced processing time for conflict detection and resolution.

**Adaptive segmentation:**

6. A distributed protocol with a name service (directory) is presented. This protocol manages incremental changes to segments such that new replicas can be established and unused ones can be removed concurrently, based on current needs for data at each individual node.

7. We introduce a generic deallocation mechanism that uses two parameters only, for the configuration of a generic sporadic expected access pattern.

8. ViFuR improves resource usage considerably, in comparison with two important alternative replication methods. We baseline ViFuR, using simulation, against two important alternative replication methods: Full replication and No replication (distributed transactions). For a typical workload, with a few on-demand replicas added, the storage used with ViFuR is somewhat higher than for no replication. Bandwidth usage for ViFuR is higher than for no replication, but remains lower than full replication even with as many clients and database nodes. This result is due to not all objects are used. The bandwidth usage for ViFuR includes communication both for establishments and for update replication, while no replication transfers transactions to other nodes and use no bandwidth for update replication. The sum of the transaction delays for ViFuR is a magnitude lower than for no replication, since many data objects are already available locally when transactions execute. Also, the growth of resource usage is clearly lower than for the two baselines, both for an increasing number of nodes and for an increasing number of database clients, such that ViFuR is obviously more scalable than the alternatives.

9. The scalability evaluation, in terms of resource usage, shows that resource usage scales with the number of replicas, as required by the workload. A set of key scale factors are varied independently: the
9.2 Application profiles for using Virtual Full Replication

In our exploration of using ViFuR for scalability and flexibility, we see that applications with certain properties benefit more than other applications.
These properties are condensed into the following list of application profiles. The profile descriptions serve as guidelines in choosing applications that benefit from using a distributed real-time database with ViFuR.

- Applications that are large-scale while resource-constrained: The application has many nodes or uses a large database, but only few objects are shared or updated frequently. Structured storage and high availability of data is needed, even when data objects are not used frequently. Data objects are typically shared as a large set shared over a few nodes, or as a small set shared over many nodes.

- Dynamic distributed applications that benefit from a low-complex data interface: Applications where the complexity of each node is reduced by using a simple transactional interface to data storage, where an assumption of full data availability at the local node simplifies complexity of computation and communication. The whiteboard communication paradigm, with shared storage accessed by transactions at the local node, is a useful communication interface that also offers scalability and fault tolerance, not available with point-to-point communication.

- Applications that publish or use certain sets of processed data, as a single processing node, or are included in a hierarchy of computational nodes: Segments are created based on the data actually shared between nodes, and different sets of data are shared by different sets of nodes, in hierarchies or in unstructured groups. Essential examples of such systems are 1) Information Fusion Infrastructures for distributed information fusion, where some more powerful nodes are assigned fusion tasks, and where fused data is published for use by a subset of all nodes and where the data users are located. 2) Embedded distributed real-time applications (such as a distributed system of ECUs in a car) that share some key data for an overall mission, while processors have local data for dedicated real-time processing and control tasks.

- Distributed systems with real-time agents traveling over a geographical area, where adaptiveness brings the data along the movement of
the agent, such that data is always locally available at the closest database node. One example is a surveillance system built as a wireless sensor network. Another example is a traffic system, where vehicles communicate with base stations in an infrastructure along the roads.

- Distributed systems with a need for real-time access of data: ViFuR provides predictable access for systems with a mix of both hard and soft transactions, where the accesses by hard transactions can be pre-specified and where soft transactions can tolerate some delays for the benefit of adaptiveness.

- Applications with a high degree of cohesive data: Large sets of cohesive data (data sharing the same property settings) are assigned to the same segment, and can be processed and migrated between nodes in blocks. At recovery of a failed node, an entire segment can easily be recovered as one consistent unit of data objects. The adaptive establishment of such entire segments, at recovery of a node or for local setup of the entire segment for availability during execution, is easily done by processing a transaction that accesses all the objects of the segment.

### 9.3 Discussion

The ViFuR schemes assume that the database usage will require a small number of replicas of data objects. For applications where the required number of replicas is large, the actual resource usage approaches the resource usage of full replication. Also, ViFuR uses additional resources for its management of data, such that ViFuR may use even more resources than full replication in total. The assumption of a required small number of replicas is valid for many systems, as we argue in this thesis. Further, even with a large number of data users, the required number of replicas may not be as high as with full replication. In Figure 6.7.(2), we show that with the number of clients equal to the number of nodes (50) ViFuR still uses less resources than what full replication uses for the same usage, since even a large number of clients do not use the entire database replica that is available at every
node.

The database replication method used by ViFuR in the thesis applies a push-replication approach, which does not consider the access frequency for the periodic usage of data. With a push/pull-aware replication method that replicates with the frequency at which updates are needed, the number of updates to be sent may be reduced and resource usage may become lower, compared to a method that always sends updates to other replicas when objects are updated. The work in the thesis focuses on availability of local replicas for local timely access of transactions, and does not consider the validity of the values of data objects, or the update frequency needed by data users. The read and write sets, as declared by transactions, is the specification of data needs. In future work, we will use the data reference management already in use by de-allocation of replicas, for the adaptation of push-replication for the periodic use of data, for push/pull replication management.

In Chapter 4 we introduce segmentation for multiple properties and multiple segmentations. In subsequent chapters, we use segmentation on allocation, and do not use multiple segmentations. Allocation is most important for local availability and a prerequisite for other properties to be used. We expect that only a few segmentations are of interest in typical applications. For instance, for the same database one segmentation can be done for allocation, and used by update propagation to send out updates to certain nodes with replicas. At the same time, a segmentation on consistency model for the data objects is used to know how to replicate updates, that is, which queue should the update be sent to (Figure 4.1). Another reason for not using other segmentations than for allocation, in subsequent chapters, is the fact that the segmentation on allocation is directly incrementally changed with adaptive ViFuR during execution, while the static pre-specified set of properties are not changed at all. Still other segmentations, such as the segmentation on consistency model, may be used by the nodes in the adaptive case. In this thesis, we have not considered adaptive changes of other segmentations than for the property of allocation.

With group-based replication, the cost of communication will be even lower than with point-to-point update replication that we assume in the eval-
9.3 Discussion

Discussion of ViFuR. Instead of sending $k$ update messages for $k$ replicas, only one message is sent. The expected improvement in resource usage is thus generally a factor of $k$. The actual number of replicas needed in a certain application is application-dependent, so the improvement with group based communication can only be known by analyzing an application. For the standard parameters used for our evaluation of ViFuR-A in Chapter 6, where there are very few replicas, group based replication is not expected to result in a much lower resource usage.

The deallocation policy used in the thesis is aimed at generic access patterns defined by a few parameters. Since the effectiveness of a de-allocation policy is closely connected to the model of the access patterns, more tailored deallocation policies could deallocate replicas more efficiently. With the generic policy used in the thesis, detection of ceased accesses is delayed and replicas are removed after a configurable timeout. For this reason, replicas are allocated longer than needed, and resources are used for no purpose. Besides having a more tailored deallocation policy, a roaming protocol could improve effectiveness. In this thesis, we have avoided such optimizations, in favor of a low-complexity specification based on the working sets of transactions’ only. Even with a simple roaming protocol in place, replica deallocation could be improved for specific scenarios: Nodes in a WSN that use ViFuR-ASN already makes use of topological information, since each node knows the neighboring nodes. A simple roaming protocol may be telling neighbors of new replicas, and the client ID used to access that new replica. All neighbors that have allocated a replica for that particular client may immediately remove the replica, without any timeout, assuming that a client has moved to a new node and only needs the new replica. For specific scenarios, other similar optimizations are likely to provide simple but effective improvements to deallocation.

To reduce blocking time for transactions waiting for replica establishments, we have proposed the use “prospective” transactions (Section 7.2.4). Such a transaction is sent to a node in advance of a time-critical access, to declare the need for the establishment of certain replicas to the node. Prospective transactions are based on the knowledge in the database client of the data needs coming. With improved knowledge about future accesses,
the application may largely reduce blocking time. Further, also the database may reduce the blocking time with more detailed knowledge about accesses coming. Database functionality that assesses the access patterns dynamically, and learns how the database is accessed, may issue prospective establishments as well. The more advanced such prediction is, the more can be gained in terms of reduced blocking time.

9.4 Future work

There are multiple ways that the work in the thesis can be extended in the future. We divide it into sections below. The ViFuR approach itself may be further explored, or ViFuR may be used to improve several areas of communication in large-scale systems.

9.4.1 Future refinements of ViFuR

- The protocol for the establishment and removal of replicas has been extensively used in many simulation rounds and for many configurations for the evaluation of ViFuR. During simulation some concurrency issues with the protocol were found and also resolved (Chapter 6.2.8). In spite of extensive execution, the protocol has not been proven for correctness. Since it is a relatively complex distributed protocol, proving correctness from a concurrency point of view is a difficult task. However, an alternative approach is to build a model of the protocol and check for correctness using a model checking tool.

- The directory used by ViFuR-A was evaluated with a single directory node only, and not as a replicated directory. To improve ViFuR-A, such usage could improve fault-tolerance as well as avoid a hot-spot and a single point of failure, that is caused by the use of a single directory only. A replicated directory could use eventual consistency and be stored in the eventually consistent database itself. The addition of new replicas of data objects is information that can be eventually consistent among replicated directory nodes. However, a replicated directory needs to guarantee that the last replica in the system cannot
be removed, thus the removal of replica allocations may need a global agreement between directory replicas. We use the term "the last replica" for the single or the number of replicas that must remain in the logical database for fault tolerance reasons.

- There are alternatives to a name service approach to find replicas of objects. A distributed hash table (DHT) is an approach that may be evaluated for finding named data objects. DHTs are widely used in large peer-to-peer networks to find named data, where there is no need for a central name service.

- The establishment delays for new replicas depend both on bandwidth availability and the concurrent transaction processing on the nodes involved during establishments. The simulation parameters used in the evaluation of ViFuR give a low utilization of both bandwidth and processing time at nodes. With the evaluation of ViFuR-ASN, the cost of establishment varies due to the delay difference caused by the difference in hop count to different recovery source nodes, such that the communication delay influences establishment time somewhat. Due to the many samples the average delay is still relatively constant. For a more deterministic delay under various conditions (possibly with a bound on the establishment time), bandwidth and processing resources need to be controlled. With regard to bandwidth, a real-time network is required such that the network ensures timeliness, or there needs to be reserved bandwidth for establishments.

- For our evaluation of ViFuR-ASN, we used fixed sensors that never change their position, while data clients are mobile. With the multi-tier database approach presented, sensors may also move in the area where database nodes can be accessed. A typical monitoring agent traveling through an area is likely to carry own sensors on the mobile platform. An approach with mobile sensors would need a way to manage movements of sensors, such that the search for sensors becomes efficient. One approach is to tag the update with the sense-location, such that the client can decide whether the updates received are still valid for the area to be monitored. If no more updates are being
received for the area of interest, a new location-directed search is needed. At the same time, the old data objects published by sensors outside of the monitored area should not be accessed anymore, and data replicas not accessed anymore need to be deallocated.

9.4.2 Segmentation for data aggregation

In addition to the scalability benefits from virtual full replication, a segmented database supports aggregation. Aggregation can be used to transform sensor data into higher-level information inside the network, before propagating it all the way to the edge nodes.

By using multiple tiers in a segmented database, hierarchies of processed data can be built up. Such refinement processing may use active rules of the database to trigger the processing of data from one segment level into results published in other segments shared by other nodes. Not only a hierarchical organization of nodes can use this kind of setup, but also systems where nodes cannot be arranged in such a structured manner. In an Information Fusion infrastructure, the database can be segmented for multiple levels of fusion, where fusion nodes read from segments shared with the lower tiers, and write into segments shared with upper tiers. In this way, a multi-tier distributed real-time database infrastructure can deliver many levels of fused data within the area covered by the distributed database.

Aggregation can be achieved by queries, such as used in Cougar (Bonnet et al. 2001) and TinyDB (Madden et al. 2005), to filter events to be propagated. This requires a query processor, typically at a base station at the network edge. We propose to trigger and execute active database rules for filtering and aggregation, distributed inside the network at the more powerful gateway nodes. Active rules define aggregation processing, and rules write the aggregated data back to the database for availability to all data users. Active rules in a database approach allow aggregation in several refining steps, both for in-network sinks and for sinks at the network edge.
9.4 Future work

9.4.3 Usage with distributed information fusion

For many applications, the fusion of information is often performed at a central location, where all information to be fused must be collected. In case the fusion is performed at a dedicated fusion node inside a network, all information must still be sent to be fused at that single node. WSNs have very unreliable communication links, and sensor readings may need to be sent redundantly to safeguard against losing sensor readings due to network loss. In our opinion, it seems an advantage to read, store and use sensor data close to the sensor, in order to reduce unreliable communication. Further, a central fusion node inside or at the edge of the network is a bottleneck and a single point of failure. In addition, all nodes using the fused information need to rely on communication links to the fusion node, and on the availability of the fusion node itself. Moreover, all information must pass the single node, which not only needs enough resources to fuse the information, but must also be able to handle the number of requests sent to and from it.

The simple filtering of updates at the source may reduce the information content of data, and subsequent fusion based on the updates becomes less effective. Also, reducing the amount of data sent is essential for scalability and for energy usage reduction. However, while the amount of data to send is reduced, fusion must not suffer from information reduction. To overcome this, fusion processing must be located close to sensors. This allows data reduction by fusion, to preserve the information in the data to be propagated, and decreases resource usage while improving the availability and fault tolerance of the information fused. In our future work, we aim to explore resource efficient in-network fusion that utilizes a distributed real-time database with ViFuR. Distributed fusion needs to be supported by an infrastructure for effective fusion, while reducing the resource usage, in order to allow fusion processing at multiple and concurrent nodes, as well as to increase reliability and scalability. In such exploration, we search for the properties of a useful infrastructure for distributed fusion. To the best of our knowledge, no available data-centric scheme uses a distributed real-time database to meet several challenges in WSNs.

Information Fusion (IF) approaches can support wireless sensor networks in improving confidence of sensor readings by fusing complementary
information. In addition, information fusion may improve the service of a WSN, such as the correlation of readings, classification for signatures, and uncertainty management. Information fusion typically makes use of multiple data sources to reduce uncertainty, for improved situation awareness. IF-approaches have been used in several WSN applications. In ALARM-NET (Wood, Virone, Doan, Cao, Selavo, Wu, Fang, He, Lin & Stankovic 2006), Dempster-Shafer’s evidential theory was used to classify sensor readings in order to associate readings with movements of people, and to create traces for several elderly individuals in an apartment. The survey paper of Nakamura (Nakamura, Loureiro & Frery 2007) includes several other examples of using Information Fusion for WSNs.

We recognize several research challenges in the work for a whiteboard approach for in-network Information Fusion, such as we propose: The allocation of fusion nodes must to be managed, particularly in a dynamic system where the needs for fused data shift over time, and where data is suddenly required at other nodes than before; Lineage management for loop avoidance to avoid erroneous reinforcement by re-using updates multiple times; Adaptive mission-oriented resource management (such as allocation of data) to meet the changing distribution and fusion needs over time; Resource management schemes need to be defined, such that resources and requests can be matched, and overload can be managed; Used data should be highly available, without consuming more resources than those available; Scalability is essential, thus search algorithms for raw and fused data in large scale systems are needed; Generic and specific infrastructures properties need to be found. Further, the need for consistency management, active functionality and scalability need to be further understood, in the context of the information fusion infrastructure needs.
References


Andler, S. F., Brohede, M., Gustavsson, S., & Mathiason, G. (2007), DeeDS NG: Architecture, design, and sample application scenario, in I. Lee,


References


annual ACM/IEEE international conference on Mobile computing and networking’, Seattle, WA USA, pp. 263–270.


References


References


References


Wu, C. (1993), Replica control protocols that guarantee high availability and low access cost, PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA.


Dissertations

Linköping Studies in Science and Technology


No 1037

No 938

No 940

No 943


No 945

No 948

No 951

No 954


No 1038

No 1041

No 1044


No 947

No 950

No 953


No 963


No 972

No 979


No 983

No 991


No 998

No 1002


No 1000

No 1005


No 1006

No 1009


No 1014

No 1015


No 1018

No 1019


No 1020

No 1023


No 1038

No 1041

No 1044


No 1050

No 1052


No 1057

No 1060


No 1061

No 1065


No 1070

No 1075


No 1079

No 1082


No 1086

No 1090


No 1095

No 1098


No 1100

No 1104


No 1107

No 1110


No 1118

No 1122


No 1127

No 1130


No 1135

No 1139


No 1142

No 1145


No 1150

No 1155


No 1160

No 1165

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
</table>

**Linköping Studies in Statistics**

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
</table>

**Linköping Studies in Information Science**

<table>
<thead>
<tr>
<th>No</th>
<th>Name</th>
<th>Title</th>
<th>Year</th>
<th>ISBN</th>
</tr>
</thead>
</table>