

# Virtual Full Replication by Adaptive Segmentation\*

Gunnar Mathiason, Sten F. Andler  
University of Skövde  
{gunnar.mathiason, sten.f.andler}@his.se

Sang H. Son  
University of Virginia  
son@cs.virginia.edu

## Abstract

We propose *Virtual Full Replication by Adaptive Segmentation (ViFuR-A)*, and evaluate its ability to maintain scalability in a replicated real-time database. With full replication and eventual consistency, transaction timeliness becomes independent of network delays for all transactions. However, full replication does not scale well, since all updates must be replicated to all nodes, also when data is needed only at a subset of the nodes. With *Virtual Full Replication that adapts to actual data needs*, resource usage can be bounded and the database can be made scalable. We propose a scheme for adaptive segmentation that detects new data needs and adapts replication. The scheme includes an architecture, a scalable protocol and a replicated directory service that together maintains scalability. We show that adaptive segmentation bounds the required storage at a significantly lower level compared to static segmentation, for a typical workload where the data needs change repeatedly. Adaptation time can be kept constant for the workload when there is sufficient resources. Also, the storage is constant with an increasing amount of nodes and linear with an increasing rate of change to data needs.

## 1 Introduction

In a distributed database, performance 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 need to be 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 on the network. In a *fully replicated* database the entire database is available at all nodes, but there are typically replicas that will never be used. Further, full replication uses excessive system resources, since the system must replicate all updates to all the nodes. This causes a scalability problem in the usage of bandwidth for replication of updates, usage of storage for data replicas, and use of processing for replicating updates and resolving conflicts for concurrent and conflicting updates.

We have developed an adaptive approach for *virtual full replication (ViFuR)*. ViFuR manages resources for scalability, by adapting to changes in data needs during execution. ViFuR reduces resource usage by avoiding irrelevant replication that blindly

sends updates to all nodes. In this paper we extend our previous approach of static segmentation based on pre-specified data needs (ViFuR-S) [12], with adaptation to new data needs during execution (ViFuR-A). A segmented database scales with the degree of required replication of data objects, and therefore it has scalable usage of resources when the required degree of replication is bounded. The required degree of replication is application dependent, but for many systems it can be assumed that only a small number of replicas are required for most of the data objects.

The use of a distributed database as a suitable infrastructure for communication in emergency management has been proposed by Tatomir and Rothkrantz [14]. It may be used as a *whiteboard* for reading and publishing current information about the state of a mission, supporting situation awareness through information collected cooperatively by all the users. With a whiteboard architecture, users need no specific addressing of recipients within the communication infrastructure. Consider rescue teams for wildfire-fighting, equipped with portable computers that can communicate. Firefighters share mission information through a distributed real-time database. Some sub-groups of firefighters may share specific information of mutual interest. The firefighters have local deadlines to consider, such as updates to status of open retreat paths, status of supplies, and location of colleagues. Local information may conflict with the global view of information received from command centers, and such conflicts must be detected and resolved. The wildfire-fighting scenario highlights the need for a large-scale distributed real-time database approach.

## 2 Background

### 2.1 The DeeDS Database Architecture

The main property of real-time systems is timeliness, which can only be achieved with predictable resource usage and with sufficiently efficient execution. To improve predictability, the distributed real-time database system DeeDS [1] stores its database entirely in main memory, to avoid disk I/O delays caused by unpredictable access times for hard drives. Also, accesses to main memory are many times faster. To avoid transaction delays due to unpredictable network delays, the database is (virtually) fully replicated to all nodes. This makes transaction timeliness independent of network delays and network partitioning, since there is no need for remote data access during transactions. In DeeDS, transactions always execute at the local node, where all objects

needed are available. A (virtually) fully replicated database with *detached replication* [6], where replication is done after transaction commit, allows *independent updates*, that is, concurrent and unsynchronized updates to replicas of the same data object. Such independent updates may cause database replicas to become inconsistent, and inconsistencies must be resolved in the replication process by a conflict detection and resolution mechanism. In DeeDS, updates are replicated detached from transaction execution, by *propagation* after transaction commit, and *integration* of replicated updates at all the other nodes. Conflicting updates are resolved at integration time. Temporary inconsistencies are allowed and guaranteed to be resolved, giving the database the property of *eventual consistency* [13]. Applications that use eventually consistent databases need to be tolerant of the temporarily inconsistent replicas, and this is the case for many distributed and embedded applications. In a (virtually) fully replicated database using detached replication, a number of predictability problems that are associated with synchronization of concurrent updates at different nodes can be avoided, such as agreement protocols or distributed locking of replicas of objects, and 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.

## 2.2 Database Model

A database maintains a finite set of logical data objects  $\mathcal{O} = \{o_0, o_1, \dots\}$ , representing database values. Object replicas are physical manifestations of logical objects. A distributed database is stored at a finite set of nodes  $\mathcal{N} = \{N_0, N_1, \dots\}$ . A replicated database contains a set of object replicas  $\mathcal{R} = \{r_0, r_1, \dots\}$ . The function  $R : \mathcal{O} \times \mathcal{N} \rightarrow \mathcal{R}$  identifies the replica  $r \in \mathcal{R}$  of a logical object  $o \in \mathcal{O}$  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  $\langle \mathcal{O}, \mathcal{R}, \mathcal{N} \rangle$ , where  $\mathcal{O}$  is the set of objects in  $D$ , and  $\mathcal{R}$  is the set of replicas of objects in  $\mathcal{O}$ , 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.  $\mathcal{N} = \{N \mid \exists r \in \mathcal{R}(\text{node}(r) = N)\}$ .

We model *transaction programs*,  $T$ , with three sets: The set of objects read by the transaction,  $\mathcal{READ}_T$  (the read set); The set of objects written by the transaction,  $\mathcal{WRITE}_T$  (the write set); and the conflict set  $\mathcal{CONFLICT}_T$ . The conflict set is a subset of updates,  $\mathcal{CONFLICT}_T \subseteq \mathcal{WRITE}_T$ , with concurrent and conflicting updates at other nodes. With this notation, a transaction program  $T$  can be defined as  $T = \langle \mathcal{READ}_T, \mathcal{WRITE}_T, \mathcal{CONFLICT}_T \rangle$ . Also, we refer to the size of the read set as  $r_T = |\mathcal{READ}_T|$ , the size of the write set as  $w_T = |\mathcal{WRITE}_T|$ , and the size of the conflict set as  $c_T = |\mathcal{CONFLICT}_T|$ . The working set  $\mathcal{WS}_T$  is the union of the read and write sets of the transaction program  $\mathcal{WS}_T = \{\mathcal{READ}_T \cup \mathcal{WRITE}_T\}$ . A *transaction instance*  $T_j$  of a transaction program is executing 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  $\text{node}(T_j) = n$ . *Virtual full replication* [11] ensures that for each transaction that reads or writes database objects at a node, there exists a local replica of the object. Formally,

$$\forall o \in \mathcal{O}, \forall T(o \in \{\mathcal{WS}_T\}) \rightarrow \exists r \in \mathcal{R}(r = R(o, \text{node}(T)))$$

## 2.3 Scalability

We choose a scalability measure that relates the amount of consumed resources to increasingly larger system parameters, *scale factors*,  $s$ , for a range of interest in a particular application setting. Scalability is achieved when a scale factor can be increased, while the consumed resources as a function of the scale factor,  $g(s)$ , do not exceed the resources that are available at the increasingly larger scale,  $f(s)$ , and where the system continues to provide service at the same level of quality. A *scalability limit* may exist, as an upper limit of the scale factor where consumed resources may exceed available resources for higher scale factors. Likewise, a *scalability threshold* may also exist, as a lower scale factor where consumed resources may exceed available resources due to overhead. Thus, a system is scalable within the range between the scalability threshold and the scalability limits, where  $g(s) \leq f(s)$ .

## 2.4 Diskless Recovery

It has been shown that *incremental diskless recovery* can recover a node in a fully replicated distributed database into a consistent database replica, without the need for any of the other working replicas to be stopped or even locked [2]. In a main-memory database, data is stored in memory pages, which is the smallest memory entity that is accessed during read or write operations. It is assumed that memory management ensures that operations have exclusively access to a memory page accessed. Diskless recovery uses *fuzzy checkpointing* and this abstraction for sequentially copying all memory pages at a node (the *recovery source*), over the network to recover a failed node (the *recovery target*). The most appropriate recovery source node is selected based on some criteria (e.g. bandwidth, link cost, communication delay). Updates for memory pages that have been sent already are logged, and the log is used to update the recovery target after the fuzzy checkpoint has completed. Updates done after the log has been sent, but before the new node is known to all nodes, are forwarded from the recovery source. Once the entire database has been copied and all subsequent updates have been forwarded, incremental recovery is complete, resulting in a consistent replica at the recovered node. In this work, we use incremental diskless recovery to setup new replicas of objects, and use database objects as the smallest entity locked during checkpointing.

## 3 Problem

As mentioned earlier, full replication at each node with data accesses being independent of network delays is highly desirable. However, a fully replicated database comes with a very high cost in resource usage. By pre-specifying actual data needs based on analysis of the transactions using database, irrelevant replication can be avoided. Under the assumption that the required degree of replication is bounded, the degree of replication is bounded and thereby the resource usage is bounded. Our initial work of segmentation based on pre-specified data needs improves scalability by scalable usage of three key resources: bandwidth, storage, and

processing. The database is divided into segments, where each segment has an individual degree of replication. Such replication makes data available only for the data needs that are known prior to execution [11] [12]. However, with this approach to scalability the database loses its flexibility to execute arbitrary transactions at arbitrary nodes, since data requirements must be static. By allowing segmentation to adapt to data needs that are not pre-specified, flexibility can be regained while scalability can be maintained. We consider three levels of dynamic changes to data needs for such flexible data access:

Case 1: An objects is added to an existing static segment that has information of where other segment replicas reside. Adding an object to an existing local segment will cause no unpredictable delay to the executing transaction, and the new object can immediately be used after it has been initiated. A global object identifier can be based on the creation node and a strictly increasing local counter. No objects are ever removed from segments.

Case 2: A missing object is detected at a node of execution, and the entire segment it belongs to is loaded to the node, since objects in the same segment are tightly related. The currently available segments and their objects are listed at each node. To load a segment, the node needs to find another node that has a replica of the segment needed.

Case 3. A missing object is loaded individually to the node where needed, rather than as an entire segment. The current segments and available objects are listed at each node. Re-segmentation may occur (in contrast to Cases 1 and 2) based on the current requirements for individual objects. Both addition and removal of objects are allowed, and re-segmentation may be needed to give the most optimal allocation, with respect to resource usage, for the currently known data needs. To load an object, the node needs to find another node that has a replica of the object needed.

To find the current allocation of a missing replica in Cases 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 paper we focus on adaptations of segmentation, to meet unspecified requirements of creation of new objects, allocations of new replicas of existing objects, and removal of replicas. This covers the more complex needs of Cases 2 and 3.

## 4 Approach

With virtual full replication every database client have an image of a fully replicated database. The database system manages knowledge of what is needed for database clients to perceive such an image, and there is typically no unnecessary replication. We consider a system with up to several hundreds of nodes where database clients publish information in the database used as a whiteboard. All accesses to the database is done by means of transactions. Global object identifiers enable distributed object sharing, and the first appearance of an identifier creates a new database object. We take: independent updates, replication, conflict detection and conflict resolution approaches of DeeDS, as described in Section 2.1, and replication uses point-to-point communication. We also assume that the network guarantees delivery, and can assign unique node identifiers when new nodes are added.

## 4.1 ViFuR-S

Our approach for Virtual Full Replication by static segmentation [12] is aiming at optimizing allocation of a bounded set of replicas to nodes in a replicated database. Based on a set of properties of each database object, the objects with identical or compatible properties are combined into segments, which have a bounded and individual degree of replication. With a bounded set of replicas, resource usage can be bounded and the database becomes scalable. A segment is represented by a pair  $\langle \mathcal{O}, \mathcal{N} \rangle$ , where  $\mathcal{O}$  is the set of data objects in the segment and  $\mathcal{N}$  is the set of nodes hosting replicas of objects in the segment.

Segmenting a database on a pre-specified set of known data references is straightforward, since a replication schema can be derived directly from the set. With additional data properties to consider apart from the required allocation, there is a combinatorial increase in the number of segments. We have presented an algorithm that handles the combinatorial problem of segmenting a database on multiple properties [12] that allow multiple segmentations, where dependencies between properties are considered. For each object, a key is generated to uniquely code data property settings, and objects are grouped into segments by this key. Consider the following example. Assume a database  $\langle \{o_1, \dots, o_6\}, \mathcal{R}, \{N_1, \dots, N_5\} \rangle$ , where seven transactions  $T_1, \dots, T_7$ , execute. Each transaction has a read set (prefixed by  $r$ ), a write set (prefixed by  $w$ ) and an execution node. We specify execution such that a transaction is associated with a tuple, specifying its read set and its write set. The conflict set is a subset of the write set, and therefore not considered. The transactions are  $T_1 : \langle r : \{o_1, o_6\}, w : \{o_6\}, N_1 \rangle$ ,  $T_2 : \langle r : \{o_1, o_6\}, w : \{o_6\}, N_4 \rangle$ ,  $T_3 : \langle r : \{o_3\}, w : \{o_5\}, N_3 \rangle$ ,  $T_4 : \langle r : \{o_5\}, w : \{o_3\}, N_2 \rangle$ ,  $T_5 : \langle r : \{o_2\}, w : \{o_2\}, N_2 \rangle$ ,  $T_6 : \langle r : \{o_2\}, w : \{o_2\}, N_5 \rangle$ ,  $T_7 : \langle r : \{o_4\}, N_3 \rangle$ . ViFuR-S creates the following four segments for allocating data to meet pre-specified requirements:  $s_1 = \langle \{o_4\}, \{N_3\} \rangle$ ,  $s_2 = \langle \{o_1, o_6\}, \{N_1, N_4\} \rangle$ ,  $s_3 = \langle \{o_2\}, \{N_2, N_5\} \rangle$ ,  $s_4 = \langle \{o_3, o_5\}, \{N_2, N_3\} \rangle$ .

### 4.1.1 Resource usage

For a scalability evaluation, we analyze the usage of three key resources: bandwidth, storage and processing. With full replication there are replicas at all nodes that receive updates from any updated node. Let  $\mathcal{T}$  be the set of transaction instances used in  $D$ . Every transaction,  $T_j$ , executed at a frequency of  $f_j$  generates updates by the size of its write set,  $w_j$ , to  $(k - 1)$  nodes, and where  $k$  is the degree of replication. Using our model, the bandwidth usage can be expressed as  $(k - 1) \sum_{T_j \in \mathcal{T}} w_j * f_j$  [messages/sec]. Assuming that the number of transactions depends on the number of nodes,  $O(n)$ , bandwidth usage is  $O(kn)$  for the virtually replicated database as compared to  $O(n^2)$  for the fully replicated database. For storage of a database with a set of segments  $\mathcal{S}$ , there are  $k_i$  replicas of each segment  $s_i \in \mathcal{S}$ . The required storage can be expressed as  $\sum_{s_i \in \mathcal{S}} (|s_i| * k_i)$  [objects]. The processing time for updates includes 1) Time for executing updates on a local node,  $L$ . 2) Propagation of updates to other nodes,  $P$ . 3) Integration of updates,  $I$ . 4) Conflict detection and resolution time,  $C$ . The overall processing time can be expressed as  $\sum_{T_j \in \mathcal{T}} f_j [Lw_j + (k - 1)\{(P + I)w_j + C\}]$ . For all three

resources we see that a bounded degree of replication,  $k$ , bounds the resource usage.

## 4.2 ViFuR-A

Our adaptive approach for Virtual Full Replication by Adaptive segmentation (ViFuR-A) includes distributed segment management and a protocol for allocation and de-allocation of replicas. The objective of ViFuR-A is to maintain scalability by resource management, and to ensure transaction timeliness by local availability. In ViFuR-A, replicas are kept locally available for database clients by considering the actual need. The key difference to database buffering and virtual memory is that other replicas are available only at peer nodes. ViFuR requires local availability for transaction timeliness, while buffering and caching approaches are used for improving performance, with optional access of data located elsewhere.

### 4.2.1 Segment management

Each node stores the segments for the local objects only, and the values of their common properties, such as the nodes the segments are allocated to. *Allocation* is the property that is a prerequisite for other properties of interest at the node, and can be changed during execution so that segments are reconfigured. The segmentation of the database is changed incrementally using *add()* and *remove()* operations that commute, and both operations execute in  $O(s+o)$ , where  $s$  is the number of segments and  $o$  is the number of objects.

The set of segments is bounded in size to  $O(o + \sum_{i=1}^p P_i * s)$ , where  $p$  is the number of properties used, and  $s$  is the number of distinct values each property may have. For allocation,  $s$  will be the number of nodes where the segment is stored. We assume that a bounded and static set of properties  $p$  are used. When transactions arrive that need to use objects not available, a *data fault* is raised, which initiates setup of missing object.

### 4.2.2 Object directory

A replicated *object directory* (or simply 'directory') stores the list of all database objects and their current allocations. Directory replicas are stored at a bounded number of nodes,  $k$ , in the system and are used for managing allocations, and no other properties. The directory uses a segmented storage that is updated incrementally by *add()* and *remove()* operations. The storage used for the directory scales with the degree of replication of the directory,  $k$ , since the directory is replicated to a bounded set of nodes, and segmentation uses the storage of  $O(o + \sum_{i=1}^S (k_i))$ . By introducing a directory, adaptation will require network communication. Even without directory communication, there will be communication for loading missing objects from other nodes, and the directory communication does not dominate the total data replica setup time. In this paper we assume that the directories are allocated to nodes considering high availability and low communication cost. Dynamic reconfiguration of directory nodes may be needed if connectivity or link quality changes, but we leave this for the future work.

At each database node there is also a *directory list* of the nodes that host a directory replica, used for addressing directory nodes.

This list is bounded in size since there is bounded number of directory nodes in the system.

### 4.2.3 Specification of data requirements

The current set of replicas that need to be available at a node is determined by the current working sets of the transaction executing at that node, and is called the *access set*,  $\mathcal{A}$ , such that  $\mathcal{A} = \cup_{i=1}^t \mathcal{WS}_i$ . If a transaction's working set or if the set of transactions change at a node, there is an *adaptation point* that changes the access set. At each adaptation point when  $\mathcal{A}$  changes, a recovery set of replicas  $\mathcal{O}_c$  needs to be added to  $\mathcal{A}$ , and a replacement set  $\mathcal{O}_p$ , that may be an empty set, needs to be removed from  $\mathcal{A}$ . The new access set will become  $\mathcal{A}' = (\mathcal{A} - \mathcal{O}_p) \cup \mathcal{O}_c$ . The storage available at a node (*the buffer size*)  $B$  must be greater than the access set, otherwise thrashing will occur, so  $|\mathcal{A}| \leq B$ .

With *data fault ratio* we mean the ratio  $\mathcal{O}_c/\mathcal{A}$  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, and this depends on the write sets of executing transactions.

A new node can be empty from start, and objects are setup when transactions execute at the node. By pre-specification, new nodes may also have objects allocated from initiation, and objects may be *pinned* to the node, to be disallowed for removal. Pinning allows critical transactions (that have a heavy penalty for missed deadlines) in an adaptive system, since there is no setup time for such objects.

## 4.3 Adaptation

### 4.3.1 Adaptation for missing data replicas

A transaction that accesses an object that has no local replica is blocked until a local replica has been setup. The setup protocol requests the current allocations from one of the directory nodes, and if such object does not exist, a first replica of a new object is created. For the objects that do have other replicas, incremental recovery is used to load a replica. The protocol timeline is shown in Figures 1 and 2, and can be outlined as follows: 1) Check local availability of objects required by the transaction. 2) If missing; ask the directory for availability. 2a) If available in directory; get current allocations. Send recovery request to selected nodes with current allocations. Recover objects by incremental recovery. As soon as a new replica is recovered, it can be used at the node (Figure 1). 2b) If not available in directory; reserve a new object ID there. Create the object at the database node. As soon as a new object is created, it can be used at the node (Figure 2). 3) Report the new allocation of the object to the directory. The directory informs the other nodes that have a replica of the object. Then, proceed with the blocked transaction that waits for the replica.

At each adaptation point, the node asks the directory for the allocation of  $|\mathcal{O}_c|$  objects. The network load for such request,  $l_{req}$  is  $l_{req} = |\mathcal{O}_c| + |\mathcal{O}_c| * k$ , where  $k$  is the bound for the degree of replication of objects. For recovery of the missing replicas, the network load is  $l_{rec} = |\mathcal{O}_c| * |o_{max}|$  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  $l_{updc} = |\mathcal{O}_c| + |\mathcal{O}_c| *$

$((k-1) + (k_d-1))$ , since  $|\mathcal{O}_c|$  bytes are first sent to the directory and then the directory updates all other nodes, using  $|\mathcal{O}_c| * (k-1)$  bytes, and all other directory nodes, using  $|\mathcal{O}_c| * (k_d-1)$  bytes. Similarly, for replaced objects the replacement load is  $l_{updp} = |\mathcal{O}_p| + |\mathcal{O}_p| * ((k-1) + (k_d-1))$  bytes. The worst case load,  $l_{max}$ , where all missing objects  $\mathcal{O}_c$  are recovered from another node, and where  $\mathcal{O}_p$  is replaced will be  $l_{max} = l_{req} + l_{rec} + l_{updc} + l_{updp}$  bytes. The adaptation time for recovery,  $t_a$ , will be  $t_a = l_{max}/H$ , where  $H$  is the available bandwidth. With  $n$  nodes adapting concurrently at a shared network, the overall adaptation time is  $T_a = t_a * n$ .

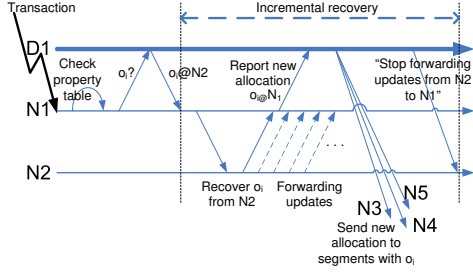


Figure 1. New allocation for object

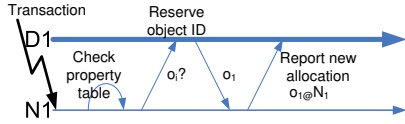


Figure 2. New object instantiation

For efficiency, objects to be retrieved from the same recovery source are combined into recovery blocks. The more objects there are to recover, the less will be the protocol overhead per retrieved object. As long as the new allocation is not known by all nodes, data updates at existing replicas need to be forwarded by the recovery source to the recovery target. To ensure that such forwarding will send all updates, the recovery source is the last node to receive the new allocation from the directory. We here assume that the underlying network guarantees the order of delivery. For networks with differentiated links and where order is not guaranteed, each message about the new allocation needs to be acknowledged, for the directory to know when other nodes are informed about the new allocation.

### 4.3.2 Removing allocations and removing objects

A removal of a replica is initiated at the local node. It can be caused by a replacement algorithm, by an explicit messages from a user transaction, or by a request from the directory. A good replacement policy will ensure that replicas with a high rate of accesses are favored. There are several suitable Least Recently Used (LRU) based policies available, such as the LIRS replacement algorithm [9]. Also, to avoid trashing the available buffer size must be larger than a certain replacement policy requires under a given workload.

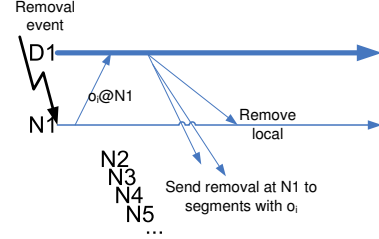


Figure 3. Removing allocation for object

Replica removal is outlined in Figure 3. There are situations when a replica can not be removed: 1) When a replica is locked by an executing transaction, the removal must be postponed until the transaction finalizes. 2) For committed transactions with updates to be propagated (after transaction commit), the replica removal need to be postponed until the propagation is done. 3) Replicas that are in use as a recovery source can not be removed until the recovery has finalized. 4) The database may have a minimum degree of replication guaranteed for objects. For the last two situations the directory needs to be available, since global constraints must be considered. A removal also need an agreement among the directory replicas. At removal, the local segments are updated later than the directory segments. For this reason, a node may still have removed nodes temporarily listed in its segments. This may cause object updates to be sent to nodes with a newly removed replica. Such updates are discarded there, and once the local segment is updated, such updates will not occur.

## 4.4 Concurrency Issues

The adaptation protocol of ViFuR-A is distributed, and every adaptation request is initiated at the local node, to be propagated to the other nodes. The add and remove operations commute and gives order independence, both for the local node and the directory. This ensures that the segment information eventually become consistent. There are two cases when concurrency is intricate: 1) The same object is recovered concurrently and independently at two nodes from different recovery sources. During the setup of a new replica, the recovery source propagates any data updates to the recovery target. With different recovery sources, different updates may reach each recovery target, which may seem to cause inconsistencies. However, DeeDS conflict detection and resolution mechanism will detect and resolve them. 2) An object is concurrently recovered from the same recovery source for two different recovery targets. The directory distributes new allocations and deallocations to existing replicas, using information from the recovery target at the time of recovery completion. Consider the example in Figure 4. The recovery at node 2 of object  $o_1$  will not know at recovery completion, that node 1 will need the new allocation once completed. Our approach is to list concurrent recoveries at the directory. At recovery completion, the content of the list is used to augment the message sent to all nodes with a replica. In the example, this list will contain " $o_1@N2$ " at recovery completion for node 1. Adding this, node 1 will know about that new allocation at node 2. For each added concurrent recovery, the list will increase.

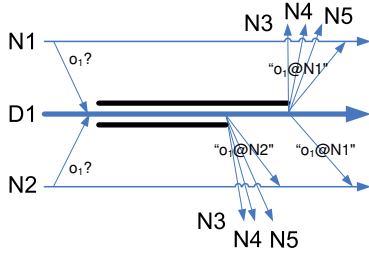


Figure 4. Concurrent recovery of object

## 5 Performance Evaluation

For performance evaluation, we simulate a virtually fully replicated distributed real-time database with adaptive segmentation, as described in Section 4. To study adaptation explicitly, we exclude detached replication of updates in the simulation. The simulator is coded in Java and executed on Java version 1.5.

We present two sets of evaluation in this section. The first set (S1) compares storage requirements for ViFuR-A compared to ViFuR-S, and we see that ViFuR-A keeps the storage requirement at a low and steady level for typical changing workload, while ViFuR-S incurs a large overhead in storage for the same changing workload. The cost for adaptation is shown as the average transaction delay, and stays at a low steady level for the same workload during adaptation periods. The second set (S2) is a scalability evaluation of ViFuR-A, by measuring storage needs and delay time for an increasing number of nodes and an increasing access change ratio. With bounded degree of replication, storage is constant for an increasing number of nodes. Storage increases close to linearly with the amount of access change; and the delay time per change request is constant with the number of nodes.

We have chosen system parameters that can be expected with a typical distributed embedded real-time system, such as a sensor network. The database uses 100 database objects per node, and the system is scaled to several hundreds of nodes. Transactions are generated for each node with a fixed number of write operations. We initiate the database by partitioning the database object indexes evenly over the nodes, and multiply indexes over multiple nodes by the degree of replication wanted. For example, 1000 objects for a 10 node database, with a replication degree of 3, gives a range of 300 object indexes to access at each node. The workload generates transactions that uses random indexes within the range. To simulate changed data needs, the range is changed periodically by a change ratio, at all nodes simultaneously. After each such change, transactions are generated for object indexes within the new range. The replicas at the node with indexes outside the new range are deallocated from the node, to simulate replaced replicas not needed anymore, and transactions are generated that causes allocations of replicas until the same level of allocation is reached, as was before the change. We choose to generate transactions such that 40% of the current range has objects allocated. For ViFuR-S entire segments are always allocated, so all objects within the initial range are allocated and this allocation remains the same throughout execution. 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 in several packages, where the transmission delay is the sum of the delay of the packages. No resource restrictions for memory and processor speeds were used, which enabled evaluation of resource usage without the effects of such resource limitations. Table 1 shows the parameters used in the simulation and workload specification.

System Parameter	Initial Value
# Nodes	10
Database Size	100 objects/node
Network delay	0.05-1.2 ms
Data size	Uniform (1-128)bytes
Workload	Value
Trans. inter arrival time	40 ms
Trans. size	5 update operations
Data fault ratio	40%
Adaptation interval	3 s
Bound on replicas required	3

Table 1. Simulation parameters

To determine the overhead, ViFuR-A was compared to full replication without segment management, using a non changing workload and with maximum allocation at all the nodes. We observe that for our standard configuration in Table 1, ViFuR-A with full allocation uses around 12-15% more storage for segment management compared to full replication, and this indicates the storage overhead for segment management. We do not include these results here due to space limitations.

### 5.1 S1: Storage needs and access delay

The first set of simulations evaluates resource usage with adaptation during 480 seconds, using 10 samples for each measurement. The static storage use of ViFuR-S can not change during execution time, so it must include replicas for all accesses. Consider a configuration of 10 nodes and 1000 database objects, each with an average size of 64 bytes, and a replication degree of 3. ViFuR-S will store 300 replicas per node, while ViFuR-A loads the replicas on demand and uses what is required by the workload.

For comparison between the schemes, we consider two extreme cases: 1) All accesses stays within these fixed 300 objects, 2) The range of the accessed object change periodically, to allow access within a changed set of 300 objects. For case 1, ViFuR-S will store all 300 objects and ViFuR-A on average 40% thereof for our workload. For the second case, ViFuR-S needs to store the entire database of 1000 objects at each node, since eventually all objects are going to be accessed. ViFuR-A will adapt and continue to use storage needed by the workload, at the cost of delays for setting up replicas.

The lowest amount of storage that ViFuR-S will ever use is with Case 1. As soon as the access set may change, ViFuR-S needs to include the union of all accesses in its static allocation.

For Case 1, ViFuR-A will store on average 40% of ViFuR-S, for the workload we use for Case 1. For case 2, the storage need for ViFuR-A remains at that level for its highest storage. We choose to measure the storage at 0%, 10%, 40%, and 60% change ratio of the accessed range, and eventually all objects of the database have been accessed. ViFuR-S requires allocation at all nodes to meet such accesses, that is 100 objects at all nodes. Figure 5.1 shows the comparison between the lowest possible storage used by ViFuR-S for Case 1, with the highest storage used by ViFuR-A for both Case 1 (0%) and 2 (10%, 40% and 60%). ViFuR-A not only adapts to the current needs but also uses less storage compared to ViFuR-S, and storage is also bounded for continuous changes, even for a high degree of changes.

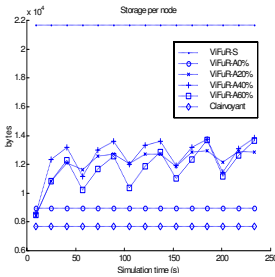


Figure 5.1. Storage

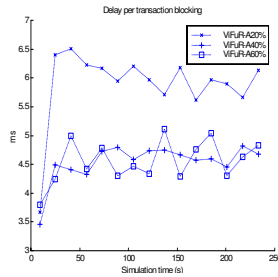


Figure 5.2. Delay

Consider for comparison an optimal adaptive system. An clairvoyant system can predict any future data requirements and can pre-load replicas to nodes before they are accessed there, without any delay at the time of the access. In such a system, the replica setup delay would be zero, and the storage would be the exactly what is needed for the number of objects pointed out by the transactions, without any overhead. For our workload it would allocate 40% of the objects in the current range, which is 7680 bytes. However, a clairvoyant system would not have the delay for setting up such an allocation. ViFuR-A uses more storage than such a system due to segment management storage. Replacement deallocates parts of the local database at each adaptation point, and the storage is reduced momentarily as can be seen in Figure 5.1. The behavior is exaggerated for higher change ratios. For the measurements, replacement at 3 second intervals interferes with the measurement point taken each 16 seconds, which explains the larger swings in the value for the 40 % and 60% change ratios. With 60 % change, the chosen workload is not enough for refilling the segments during the period, so the average fill level is lower.

Transactions are blocked until required replicas become available at the node, and for each 16 seconds we measure the summed delay time for blocking and divide by the blocking count, giving the average delay time for transactions during replica setup (Figure 5.2). For a 10% change ratio, the delay is around 6 ms, and for 40% and 60% change ratio, the delay is around 4.5 ms. The delay for more change ratios is studied in simulation 2, where we evaluate scalability. If there is an increasing resource usage for a stable workload, the delays would increase, since delay time depends on the usage of both network and processing resources.

## 5.2 S2: Storage and delay scalability

The second set of simulations evaluates resource usage with an increasing number of nodes and also higher change ratios. We use 10, 20, 50, 100, 200 and 300 nodes, and 20%, 30%, 40%, 50% and 60% change ratios. For each of the combination, we measure storage and average delay time per transaction blocked, during a period of 60 seconds, with 2 samples. Due to the bounded degree of replication, the storage is expected to remain constant with an increasing number of nodes.

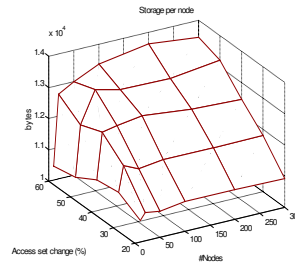


Figure 6.1. Storage

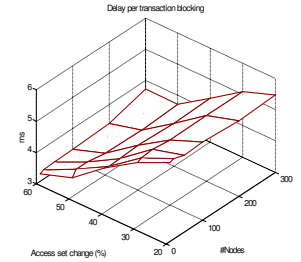


Figure 6.2. Delay

Figure 6.1 shows that the used storage is constant when the number of nodes is increased. Along with the change of the access set, the requested degree of replication remains at 3 replicas, due to our workload. The storage increases with the amount of changes to process in the system, and the increase is close to linear with increased change ratio. In Figure 6.2, we see that the total delay time per blocked transaction is close to constant with the number of nodes, since number of replicas does not increase. We also see that the delay time for each request decreases with an increasing change ratio of the access set, since more requests are combined into larger blocks of replicas to setup. At higher change requests, the adaptation algorithm takes more advantage of coordinated setup of multiple replicas simultaneously.

## 6 Related Work

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

It is a NP complete problem to find an optimal allocation of partitions of a database for minimizing the communication cost, but several heuristics-based approaches exist, using a-priori knowledge of accesses. Wolfson et al. [15] propose the ADR algorithm for replicated databases that optimizes allocation over time, by adjusting each data objects' replication tree, but this approach does not explicitly consider real-time requirements or scalability. Lin and Veeravalli [10] propose the DWM algorithm for minimizing 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, we use commuting operations and a distributed directory to avoid this. DWM is evaluated by analysis, and absolute delay time and real-time requirements are not considered. Group management and group communication enables scalable replication when group sizes are relatively small [8]. This is orthogonal to our work and will likely improve resource usage for replication in ViFuR-A. Also, segmentation may be used for group management, and we regarded this as future work.

ViFuR-A has some resemblance with database buffering, virtual memory and cache coherence approaches. Effelsberg and Haerder outline the differences between virtual memory paging and database buffer management in that locality of reference patterns are different [5], and as a consequence reference management and replacement need to be different. For buffer replacement in real-time databases, Carey et al. [3] introduces priorities for traditional Least Recently Used (LRU) schemes. The work shows the importance of admission control and scheduling of CPU and disks. Huang and Stankovic find that conflict resolution is a key factor for performance with hot data [7]. 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. [4] 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. There are many replacement approaches available for specific access patterns based on LRU. The LIRS algorithm [9] is a generic, simple and effective LRU-based approach that uses recency of accesses with reference counting, and this can be used for replacements in ViFuR-A in the future work.

## 7 Conclusions and Future Work

This paper describes a novel scheme for resource management in real-time databases. We have developed virtual full replication with adaptive segmentation (ViFuR-A), to adapt replication over time to actual data needs of database clients. The data objects used are allocated locally, to make transactions independent of network delays. We present an architecture and an adaptation protocol that uses a replicated directory service to bound the storage required for management of the global object allocation scheme, and for scalable segment management at each database node. Virtual Full Replication gives the database client an image of a fully replicated database where all objects are locally available. The client can access all database objects locally by timely transactions, and ignore issues such as object location and synchronization of updates. We show by simulation that virtual full replication by adaptive segmentation maintains scalability and preserves transaction timeliness, while adapting to new data needs.

Virtual full replication and scalable resource usage may be particularly useful in sensor networks, where partitioning is common and where nodes have limited resources. It may also be useful for renewing the sensor network by deployment of new nodes, which can request new replicas of existing objects for data migration to the new nodes.

We plan to further reduce setup waiting time for new replicas,

by examine how scheduling of replica setup may be used. With the current scheme, the directory replies to all requests in the order of arrival, and scheduling based on size of the data or the priority of transactions may further lower setup time. We also plan to examine how admission control can help to avoid unnecessary blocking of transactions waiting for replicas. Admission control may let the database client application decide alternative execution to submitting transaction, which may be blocked when the system is highly loaded with ongoing replica setup. We also plan to use a more elaborate and generic replacement policy, to increase the availability of replicas, in particular for periodic transactions. Further, with pattern detection, there is an opportunity to pre-fetch replicas for transactions that arrives with a certain pattern.

## References

- [1] S. Andler, J. Hansson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Efring. DeeDS towards a distributed and active real-time database system. *SIGMOD Record*, 25(1):38–40, 1996.
- [2] S. F. Andler, E. Leifsson, and J. Mellin. Diskless real-time database recovery in distributed systems. In *Real-Time Systems Symposium (WiP)*, 1999.
- [3] M. J. Carey, R. Jauhari, and M. Livny. Priority in dbms resource scheduling. In *Proceedings of the 15th VLDB*, pages 397–410, 1989.
- [4] A. Datta, S. Mukherjee, and I. R. Viguier. Buffer management in real-time active database systems. *The Journal of Systems and Software*, 42(3):227–246, 1998.
- [5] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM TODS*, 9(4):560–595, 1984.
- [6] S. Gustavsson and S. Andler. Continuous consistency management in distributed real-time databases with multiple writers of replicated data. In *Proc. WPDRTS*, 2005.
- [7] J. Huang and J. A. Stankovic. Buffer management in real-time databases. Technical Report UM-CS-1990-065, Univ. of Mass., 1990.
- [8] W. Jia, J. Kaiser, and E. Nett. A fault-tolerant efficient group communication protocol. *IEEE Micro*, 16(2):59–67, 1996.
- [9] S. Jiang and X. Zhang. Making LRU friendly to weak locality workloads: A novel replacement algorithm to improve buffer cache performance. *IEEE TOC*, 54(8):939–952, 2005.
- [10] W. J. Lin and B. Veeravalli. A dynamic object allocation and replication algorithm for distributed systems with centralized control. *Int. J. Comput. Appl.*, 28(1):26–34, 2006.
- [11] G. Mathiason and S. F. Andler. Virtual full replication: Achieving scalability in distributed real-time main-memory systems. In *Proc. 15th ECRTS (WiP)*, pages 33–36, 2003.
- [12] G. Mathiason, S. F. Andler, and D. Jagszent. Virtual full replication by static segmentation for multiple properties of data objects. In *Proceedings of RTIS '05*, pages 11–18, 2005.
- [13] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 2005.
- [14] B. Tatomir and L. Rothkrantz. Crisis management using mobile ad-hoc wireless networks. In *Proc. of ISCRAM 2005*, pages 147–149, 2005.
- [15] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM TODS*, 22(2):255–314, 1997.