

ORDER: A Dynamic Replication Algorithm for Periodic Transactions in Distributed Real-Time Databases ^{*}

Yuan Wei Andrew A. Aslinger Sang H. Son John A. Stankovic

Department of Computer Science
University of Virginia
Charlottesville, Virginia, 22904-4740
U.S.A.

E-mail: {yw3f, aaa7u, son, stankovic}@cs.virginia.edu

Abstract. Many real-time applications need data services in distributed environments. However, providing such data services is a challenging task due to long remote data accessing delays and stringent time requirements of real-time transactions. Replication can help distributed real-time database systems meet the stringent time requirements of application transactions. In this paper, we present two dynamic replication control algorithms designed for medium and large scale distributed real-time database systems. With the data needs information from incoming transactions, our algorithms dynamically determine where and how often the replicas are replicated. In our algorithms, the data replicas are dynamically created upon the requests by the incoming transactions and their update frequencies are determined by the data freshness requirements of these transactions. A detailed simulation study shows that our algorithms can greatly improve the system performance compared to the systems without replication or systems with simple replication strategies such as full replication.

1 Introduction

Many real-time applications need to share data that are distributed among multiple sites. For example, in naval combat control systems, sensor values are collected and shared by real-time database systems scattered throughout different combat ships and submarines[1]. In such a distributed real-time database system, remote data access consists of multi-hop network operations and takes substantially more time than the local data accesses. This potentially leads to large number of transaction deadline misses. Another problem is that due to the long remote data access time, by the time a transaction gets all the data it needs, some of its data items may have already become stale.

^{*} This work was supported, in part, by NSF grants IIS-0208758, CCR-0329609 and CCR-0098269.

Replication is an effective method to solve the aforementioned problems. By replicating temporal data items, instead of initiating remote data access requests, transactions that need to read remote data can now access the locally available replicas. This helps transactions meet their time and data freshness requirements. There are multiple ways to handle the replication control. Different replication schemes are better suited for different data workloads and database specifications. Our previous work on replication [2] works well in small-scale distributed real-time database system. In that work, full replication is used within the system. In this paper, we examine several methods of replication control in medium-scale or large-scale distributed real-time database systems and present a replication algorithm called *On-demand Real-time DEcentralized Replication* (ORDER). The ORDER algorithm is designed to work in a environment where all data types and relations in the system are known a priori and transactions are short-term periodic transactions. When a transaction arrives, it declares its period, data needs, execution time and deadline. With this information, the algorithm decides where and how often the replicas are updated. The simulation results show that this type of on-demand replication algorithm can greatly improve the system performance. In term of scalability, our algorithm can be enhanced to a replication algorithm called *On-demand Real-time Decentralized Replication with Replica Sharing* (ORDER-RS). With this algorithm, large distributed systems are divided into small groups called *cliques* based on the network topology. The replicas within one clique are shared by the clique members. Compared with the ORDER algorithm, the ORDER-RS algorithm can further improve the system performance when the system size scales to very large sizes.

The rest of this paper is organized as follows. Section 2 describes the background of replication and the related work. The system model is described in section 3. The ORDER algorithm and ORDER-RS algorithm are presented in sections 4 and 5, respectively. Section 6 presents the results of our performance evaluation. And finally, section 7 concludes the paper and discusses the future work.

2 Background and Related Work

Replication algorithms can be divided into *full replication* algorithms and *partial replication* algorithms. Full replication is a replication strategy that replicates all data to all sites in the distributed system. The benefit of full replication is that all data are available to read locally. However, full replication may not be efficient in middle-scale or large-scale distributed systems in which tens or hundreds of databases maintain thousands of data items. In those systems, the cost of maintaining replicas for all data, especially sensor data, is too high. Instead, partial replication is better suited for middle-scale or large-scale systems. With partial replication, only a portion of data items in the database are replicated and if a data item is replicated, it does not need to be replicated to all sites. Replication algorithms can also be divided into *static* replication algorithms and *dynamic* replication algorithms. When a replication is static, the location and

the number of the replicas are fixed. When a replication algorithm is dynamic, the location and the number of the replicas can change dynamically according to the transaction data needs and system conditions.

There have been a number of research papers about data replication in traditional database systems [3][4][5][6][7][8]. In recent years, many research papers in real-time databases have been published [9][10][11][12]. However, there are not many replication control algorithms for real-time database systems. MIRROR (Managing Isolation in Replicated Real-Time Object Repositories) [13] is a concurrency control algorithm for replicated real-time database systems. In the paper, a state-conscious priority blocking mechanism is added to O2PL to improve the system performance. Main-memory residence and full replication are used in [14] to ensure predictable (and available) operation in the local node. For replication control, the concept of "virtual full replication" was introduced in [15] to address scalability issues. In the paper, virtual full replication is provided by segmenting the database and allowing different segments to have different degrees of replication. This technique could be combined with our dynamic replication scheme. In our previous research[2], we proposed a QoS management algorithm for small-scale replicated real-time database systems. In [16], Peddi et. al. present a replication algorithm called *Just-In-Time Real-Time Replication* (JITRTR), which creates replication transactions based on client's data requirements in a distributed real-time object-oriented database. The JITRTR algorithm works in a static environment in which data requirement are known a priori. The JITRTR analyzes the requirements of the clients and creates transactions that make the data available. The algorithm guarantees that the data is available when needed and only valid data is read by the transactions. Compared to JITRTR, which deals with static real-time databases, our algorithms focus on a dynamic system where transactions and their data needs are not known a priori.

3 System Model

In this paper, we study a distributed real-time database system which consists of a group of main memory real-time databases connected by high-speed networks. Main memory databases have been increasingly used for real-time data management because of the high performance of memory accesses and the decreasing main memory cost [17]. In the system, a firm real-time database model is used. Tardy transactions (transactions that have missed their deadlines) are aborted.

3.1 Data Model

In our system model, data objects are divided into two types, namely, *temporal data* and *non-temporal data*. Temporal data are the sensor data from physical world. For example, in a submarine control system, they could be ship maneuvering data (such as position, speed and depth). Temporal data objects have *validity intervals* and are updated periodically. Non-temporal data do not change

dynamically with time. Thus they do not have validity intervals and they do not need to be updated by periodic system updates.

In our distributed real-time database system model, a database is called a *site*. Each site hosts a set of temporal data objects and non-temporal objects. The site is called the *primary site* for those data objects. Each site also maintains a set of replicas of temporal data objects hosted by other nodes. Fresh values of temporal data objects are periodically submitted from sensors to their primary sites and propagated to the replicas. For a specific data item, the data copy at the primary site is called *primary copy* and the copies that are replicated are called *replicated copies* or *replicas*. In this paper, we focus on temporal data services and only temporal data are replicated. However, both types of data items are accessed by transactions.

3.2 Transaction Model and Scheduling

The transactions in the system are divided into two types, *system update transactions* and *user transactions*. System update transactions include temporal data (sensor data) update transactions and replica update transactions. User transactions are queries or updates from applications. In this paper, we assume that transactions should access fresh temporal data. Based on previous research results [18], temporal data update transactions should get scheduled first, and then replica update transactions. Incoming application transactions get scheduled after temporal data update and replica update transactions.

Transactions are represented as a sequence of *operations* on the data objects. Operations of one transaction are executed in *sequential* fashion. One operation can not be executed unless all previous operations are finished. Once all operations of one transaction are finished, the transaction enters *validation stage*. At the validation stage, the system checks the freshness of the accessed data. If the accessed data items are not fresh anymore, the transaction is restarted. After the validation stage, the transaction enters the commit stage which writes log and commits.

If a transaction needs to *read* a data item at another site, it checks whether there is fresh local copy available. If there is a fresh replica locally, the transaction just reads the local copy. If the operation is not read or there is no fresh local copy available, the transaction sends out data service requests to set up cohorts at remote sites. The cohorts read or write data items on behalf of the transaction. At the validation stage, the transaction sends out a commit preparation message to all cohorts. As in the 2-phase commit protocol, the transaction commits if and only if all cohorts agree to commit.

4 ORDER Dynamic Replication Algorithm

The goal of the ORDER algorithm is to gain efficiency over replication strategies such as full replication by dynamically changing the update frequency and update duration of replicas.

4.1 Update Frequency and Duration Definitions

Before we describe the algorithm, we need to define some terms used to describe the algorithm. In our database model, each node may contain multiple temporal data items and replicas. The primary copy of a data item is updated periodically at a given *basic update frequency* (BUF) while its replicas are updated at different *extended update frequencies* (EUF) specified by the incoming application transactions. All replicas of a particular data item are updated using the fresh value from their primary copy. Therefore, the EUF upper bound for a given data item is the BUF of the primary copy. When a replica is updated periodically, it is called an *active replica*. Otherwise, it is called a *dormant replica*. An active replica will become dormant if it is no longer needed by any incoming application transactions. The time that it turns into a dormant replica is called its *closing time* (CT).

4.2 Definite Periodic Workload Model

Our paper deal with the periodic transaction workload model where transactions are periodic with definite data requirements and service durations. A transaction may arrive at any time and each transaction may contain requests for multiple data objects from different sites. The specification for a transaction is as follows:

$$\{TID, TD, EXETime, SF, DS\}$$

Each transaction specification contains a *transaction identifier* (TID), a *transaction duration* (TD), *execution time* (EXETime), *slack factor* (SF), and a *data set* (DS). The a transaction's data set consists of elements of the following format:

$$DataObject\{SiteID, DataType, DataID, FR\}$$

The specification for one data object consists of information of the *database identifier* (site ID), *data type*, *data identifier* (data ID) and the *freshness requirement* (FR) of that data item.

When a transaction arrives, it requests a data service with specific data freshness requirements and indicates the duration of the service. When this duration expires, the transaction no longer requests data objects at this site.

4.3 Algorithm Description

In the ORDER algorithm, the update frequency and update duration of replicas are dynamically controlled to satisfy the data freshness requirements of the incoming transactions. The algorithm, when receiving a transaction, evaluates the data needs of the incoming transaction and creates data replicas for the transaction if the transaction can be admitted based on the current system conditions. It also registers the update frequency and duration to the primary sites of these replicas. In the algorithm, it is the job of the receiving site to register active

replicas to their primary sites. It is the duty of the primary site to push updates to the active replicas at the extended frequencies requested by the incoming transactions.

When a local site admits a transaction, the algorithm calculates the proper update frequency and update duration for each remote data item specified by the transaction. Suppose the algorithm receives a request for remote temporal data item i from site x . The pseudo code for the replication algorithm is given in Fig. 1.

As shown in Fig. 1, when a new transaction arrives, for each remote temporal data item the transaction requests, the algorithm tests whether there is an existing active replica for the remote data item. If there is already an active replica, the algorithm compares the current update frequency of the replica with the update frequency requested by the new transaction. If the new transaction requests the replica to be updated at a higher frequency, the update frequency of the active replica is changed to the new update frequency requested by the transaction. In that case, the closing time of the replica is also changed to the current time plus the duration of the new transaction. If the replica update frequency requested by the new transaction is less than the original update frequency, the algorithm does not need to do anything because the current update frequency is high enough. If there is no active replica for the remote temporal data item, the algorithm creates an active replica for that data item using the update frequency and duration of the new transaction.

To maintain the minimum update frequency for all active replicas, the algorithm must keep track of all transactions that use the data replicas until they expire. The algorithm also needs to re-calculate the update frequency of replicas that are accessed by an expiring transaction. For example, consider the case of a transaction that requests a remote temporal data replica to be updated every 5 seconds for a duration of 100 seconds. After that, another transaction arrives at 20th second requesting the same data object to be updated every 2 seconds for a duration of 20 seconds. With these two transactions, the algorithm sets the replica update frequency to once every 2 seconds for a duration of 20 seconds. When the second transaction expires at the 40th second, the system must restore the update frequency to once every 5 seconds. The pseudo code for transaction departure is shown in Fig. 2.

As shown in Fig. 2, when a transaction is exiting the system, its requested update frequencies on active replicas are checked. If it requests the highest update frequency for an active replica, the update frequency and closing time of the active replica need to be re-calculated. The system needs to find the maximum requested update frequency without the expiring transaction. The closing time of the replica is then set to the expiring time of the transaction that requests the highest update frequency.

```

//Current_EUF(i, x): The Current extended update
//frequency for temporal data i from site x.
//New_EUF(i, x): The new transaction requested
//update frequency for temporal data i from site x.
//Current_CT(i, x): The current closing time of the
//Replica for temporal data i from site x.
//TD: The incoming transaction requested duration.

If( ExistActiveReplica(i, x) ) {
  //There exist an active replica
  if(Current_EUF(i, x) >= New_EUF(i, x)) {
    //Use Current EUF
    //Nothing to do here.
  }
  else {
    //Use the New_EUF
    Current_EUF(i, x) = New_EUF(i, x);
    //Set new replica closing time
    Current_CT(i, x) = Current_Time + TD;

    Register_Active_Replica (Current_EUF(i, x),
    Current_CT(i, x));
  }
}
else {
  //There is no active replica
  Create_Active_Replica (i, x);
  Current_EUF(i, x) = New_EUF(i, x);
  Current_CT(i, x) = Current_Time + TD;

  Register_Active_Replica (Current_EUF(i, x),
  Current_CT(i, x));
}

```

Fig. 1. Pseudo Code for Update Frequency and Duration Calculation (on transaction arrival)

```

//Current_EUF(i, x): The Current extended update
//frequency for temporal data i from site x.
//EUF(i, x): The expiring transaction requested
//update frequency for temporal data i from site x.
//Current_CT(i, x): The current closing time of the
//replica for temporal data i from site x.
//Find_Max_EUF(i, x): Find the maximum requested
//EUF for temporal data i from site x.
//Find_TD() : Find the transaction expiring time
//corresponding to the requested update frequency.

If( ExistActiveReplica(i, x) ) {
  //There exist an active replica
  if(Current_EUF(i, x) >EUF(i, x)) {
    //Nothing to do here.
  }
  else {
    //Use the maximum requested EUF
    Current_EUF(i, x) = Find_Max_EUF(i, x);
    //Set the replica closing time
    Current_CT(i, x) = Find_TD(Current_EUF(i, x));

    Register_Active_Replica (Current_EUF(i, x),
      Current_CT(i, x));
  }
}

```

Fig. 2. Pseudo Code for Update Frequency and Duration Calculation (on transaction departure)

5 ORDER-RS Dynamic Replication Algorithm

The ORDER algorithm may not perform well in large-scale distributed real-time databases. First, in large-scale distributed systems, it might be very costly or impossible for every site in the system to maintain detailed information about all data items in the system. Second, sometimes it may not be efficient to fetch all fresh data items directly from their primary site. Instead, the sit can get fresh copies from some existing active replicas that are closer, i.e., the replicas that could be reached with less transmission delays. An example is given in Fig. 3. In the figure, two real-time databases DB1 and DB2 are connected by high speed LAN. DB1 has an active replica for a remote temporal data item. The active replica is updated periodically using the fresh data values from the primary site. Now, DB2 admits a new transaction which needs the same remote data. As we can see, it is not efficient for DB2 to fetch the data value from remote site again since there is already an active replica in DB1, which can be easily reached. Fetching fresh data directly from the primary site unnecessarily increases the workload of the network and the primary site. Instead, it is more desirable to fetch the data from DB1. If the update frequency of the active replica at DB1 can satisfy the requirements of the new transaction in DB2, the system only needs to replicate data from active replica 1 to active replica 2. If the new transaction demands higher update frequency, the system can now stop replicating data from primary copy to active replica 1. Instead, it replicates data directly from the primary copy to replica 2 (at higher update frequency) and then replicates the data from replica 2 to replica 1.

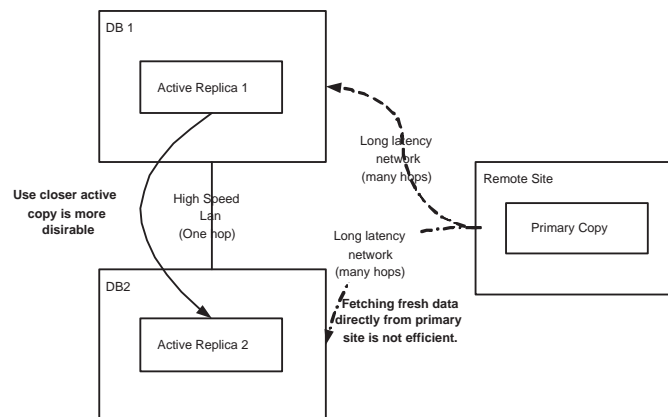


Fig. 3. Fetching Fresh Data from Closer Active Replicas

To use the idea described above, we extend our algorithm with a mechanism that shares active replicas among those sites that are close (in transmission

latency) to each other. We call this new algorithm *On-demand Real-time Decentralized Replication with Replica Sharing* (ORDER-RS). An example is shown in Fig. 4. In the figure, there is a medium-scale distributed real-time database system which consists of 24 real-time database servers. The real-time database servers are connected by 8 transmission stations. The system are divided into 6 *cliques* based on the network topology. The real-time databases within each clique are connected by wired high speed networks. While the cliques are connected by multiple-hop wireless networks. We also assume that the clique members from the same clique know the existence of each other. These assumptions match the real system configuration in combat control systems. In those systems, the real-time database servers within one ship or submarine are connected by high speed ethernet[19] while the communication between ships are via global wireless networks or on-shore transmission stations [1].

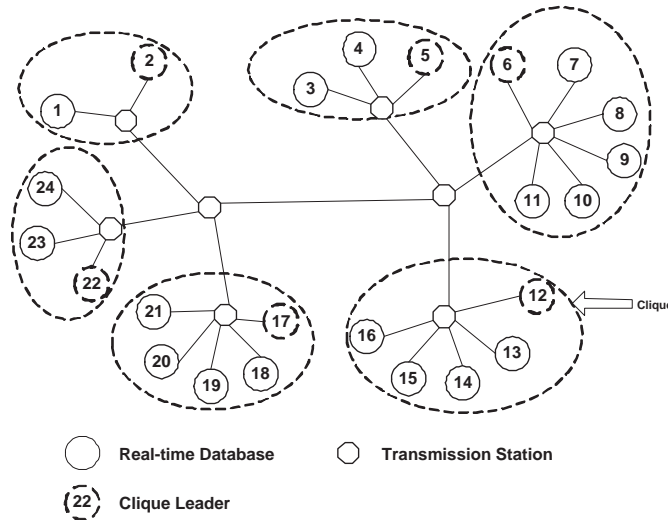


Fig. 4. Replica Sharing in Large Distributed Systems

With the ORDER-RS algorithm, the replicas in one clique are shared by the database servers in that clique. In each clique, there is a clique leader which manages the replication process in that clique. The clique leader controls all the replicas in that clique and acts as a proxy for clique members when they need temporal data items from database servers in other cliques. When a clique member needs remote temporal data, it first checks where the requested data item resides. If the requested data item resides within the same clique, it just sends a request to the clique member that has the requested data to set up the replication process. If the data resides in a database server in a different clique, it sends the request to the local clique leader. Upon receiving the request, the local

clique leader checks whether there is already a replica for the same data within the clique. If there is, the clique leader changes the replica update frequency to the current highest requested frequency and all requests from within the clique can now share the replica. If there is no replica corresponding to that requested remote data item, the clique leader sets up a new replica with the requested update frequency and duration. The update frequency and duration calculation processes on transaction arrival and departure are almost the same as those described in Fig. 1 and Fig. 2 except that the replication between two clique members within a clique is handled directly by the clique members themselves while the replication between database servers in different cliques is handled by the clique leaders.

The clique leaders can be defined statically or be selected dynamically at run time based on the system workload. For simplicity, in this paper, the clique leaders are statically determined and do not change at run time.

6 Performance Evaluation

6.1 Simulation Setting

Parameter	Value
Node #	10
Network Delay	0.05 - 1.2 ms
Temp Data #	500/Node
Temp Data Size	Uniform(1-128)bytes
Temp Data Base Update Frequency	Uniform(0.1 - 1) sec
Non-Temp Data #	10,000/Node
Non-Temp Data Size	Uniform(1-1024) bytes
Replica Creation and Deletion Overhead	0.5 ms

Table 1. System Parameter Settings

For our simulations, we have chosen system parameter values that are typical of today’s technology capabilities, e.g., network delays and replica creation overhead. The general system parameter settings are given in Table 1. The network delays are modelled by calculating the end-to-end transmission delay for each packet. Depending on the packet size (64 - 1500 bytes for Ethernet), the end-to-end delay ranges from 50 microseconds to 1.2 milliseconds. If the data size exceeds one packet size, the data is put into separate packets and the transmission delay is the sum of delays for those packets. In the simulation, the overhead of running the replication control algorithm is modelled by collecting

Parameter	Value
Transaction Period	0.5 - 2 sec
Transaction Duration	20 sec
Operation Time	0.1 - 1 ms
Temp Data OP #	1 - 8 /Tran
Non-temp Data OP #	0 - 2 /Tran
Transaction SF	5
Temp Data Access Skew	10%
Non-Temp Data Access Skew	10%
Non-Temp Data Update Ratio	10%
Remote Data Ratio	20%
Freshness Requirement	1 - 3 BUF

Table 2. User Transaction Workload Parameter Settings

the overheads for creating and deleting replicas. Each of these operations takes 0.5 millisecond.

The settings for the user transaction workload are given in Table 2. A user transaction consists of operations on both temporal data objects and non-temporal data objects. The execution time for one operation is between 100 microseconds to 1000 microseconds. The slack factor of transactions is set to 5. To model the transaction access patterns, we introduce *Temporal Data Access Skew* and *Non-temporal Data Access Skew*. A 10% access skew means that 90 percent of all transaction operations access 10 percent of that type of data objects. The *Remote Data Ratio* is the ratio of the number of remote data operations (operations that access data hosted by other sites) to that of all data operations. The remote data ratio is set to 20%, which means 20 percent of transaction operations are remote data operations. At each node, the user transaction workload consists of many periodic transactions and the average arrival rate is shown in the simulation results. All simulation results are based on at least ten runs and the confidence intervals are less than 10% of the mean values.

To evaluate our algorithm, we use *no replication* and *full replication* as two baseline protocols. These two algorithms are the simplest, but widely used replication control strategies. The transaction miss ratios and system utilizations of the three algorithms are shown in Fig. 5. As we can see from the figure, among the three algorithms, the ORDER algorithm gives the best transaction miss ratios under different transaction workloads. For example, when the transaction arrival rate is at 80 transactions per second, the miss ratio of the ORDER algorithm is below 2% while the miss ratios of the other two algorithms are around 8%. Note that the CPU utilization of the system without replication is higher than that of the system running ORDER algorithm when the transaction arrival rate is less than 140 transactions per second. Beyond that, the trend is reversed. The reason is that, at low system workload, the system without replication invokes a remote data operation each time there is a remote data access request, which tends to consume more CPU time than the ORDER algorithm. However, when the sys-

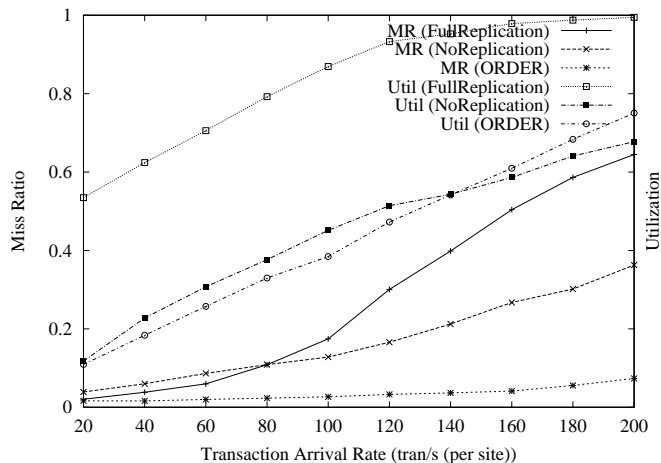


Fig. 5. Transaction Miss Ratios and System Utilizations

tem workload is high, many transactions in the system without replication miss their deadlines while waiting for remote data accesses. The CPU utilization is relatively lower because many transactions never get fully executed.

6.2 Performance with Different Remote Data Ratios

The remote data ratio of transactions has a major impact on the performance of the replication control algorithms. In this set of simulations, we increase the remote data ratio from 20% to 50% and compare the performance of three algorithms. The simulation results are shown in Fig. 6. As we can see in the figure, the remote data ratio increase affects all three algorithms. Their miss ratios all increase as the remote data ratio increases. However, unlike the no-replication algorithm, the ORDER algorithm and the full-replication algorithm are not seriously affected by the increase of remote data ratio. The miss ratio of no-replication algorithm dramatically increases as the remote data ratio increases. Note that, with 50% remote data ratio, the system without replication can not deliver acceptable miss ratio even when the system workload is very low. For example, the transaction miss ratio of the system without replication is already 8% at 20 transactions per second.

6.3 Performance with Different Parameter Settings

We also vary some other system parameters. In this set of simulations, we decrease the average transaction duration and relax the transaction freshness requirements. As shown in Fig. 7, when we decrease the average transaction duration, the miss ratio of the ORDER algorithm increases due to the increased

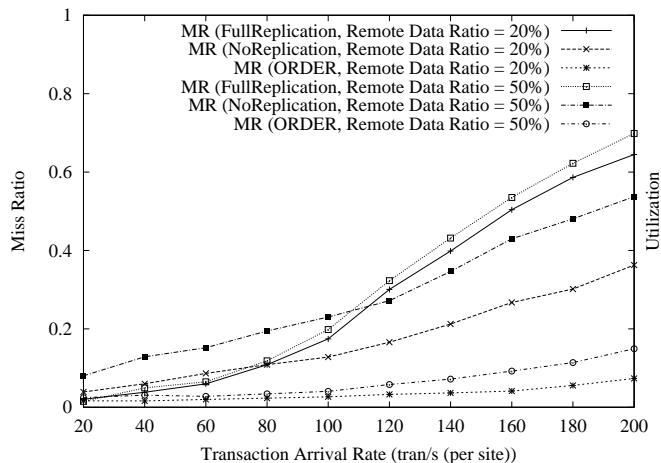


Fig. 6. Performance with More Remote Data

replica creation and deletion overhead. When we relax the transaction freshness requirements from 1 - 3 *base update frequency* (BUF) to 3 - 5 BUF, the miss ratios are improved. There is not much miss ratio improvement in absolute terms because the transaction miss ratio is already fairly low (5%).

6.4 Performance with ORDER-RS Algorithm

In this set of simulations, we simulate a larger distributed real-time database system running the ORDER-RS algorithm. The system is set up using the parameters shown in Table. 3. The system consists of 32 real-time database servers. The system is divided into 8 cliques and each clique has 4 sites. The network transmission delays within a clique and between cliques are modelled separately. The transmission within a clique is faster, which takes 0.5 millisecond to finish, while the transmission between different cliques takes as long as 2 milliseconds. These system settings can be mapped to a distributed real-time database within a naval fleet where ships and submarines share the real-time data during a combat. The simulation results are shown in Fig. 8.

As shown in Fig. 8, the ORDER and ORDER-RS algorithm perform much better than the no-replication algorithm. In fact, the system without replication can not deliver acceptable system performance at all. For example, the transaction miss ratio is as high as 30% when the average transaction workload is 20 transactions per second. It is due to the high latency of remote data access. Transactions with short execution time and short slack time can never meet their deadlines in a system without replication. Both ORDER and ORDER-RS algorithm show acceptable transaction miss ratios. The ORDER-RS algorithm shows slightly better performance in different transaction workloads due to the replica sharing, which reduces the workload at the primary sites. The effect of

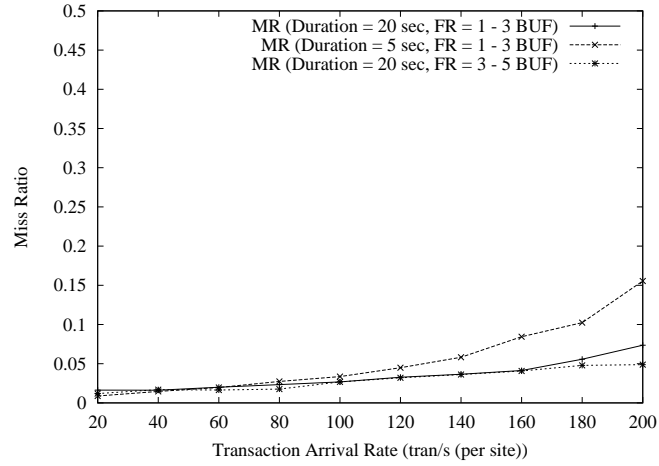


Fig. 7. Performance with Different Transaction Parameters

Parameter	Value
Node #	32
Clique Size	4
Clique Number	8
Transmission Delay within a Clique	0.5 ms
Transmission Delay Between Cliques	2 ms

Table 3. Large-Scale Distributed Real-Time Database Settings

replica sharing become more evident when the system workload gets higher. For example, the miss ratio difference between the ORDER and ORDER-RS algorithm is 2% when transaction workload is at 80 transactions per second; at 200 transactions per second, the miss ratio difference is around 8%. The reason is that when the transaction workload is higher, there are more replicas that can be shared. Another advantage of both algorithms is that the network usage gets dramatically decreased. The network usage of the ORDER algorithm is around half of that of no-replication algorithm. Compared to the ORDER algorithm, the ORDER-RS algorithm further reduces the network usage between cliques. This is particularly valuable in naval ship command-and-control applications in which the wireless transmission between two ships (cliques) is very expensive. We do not show the performance of the full replication algorithm here because full replication is not applicable to systems of this scale. The replica update workload is so high that the database servers will be overloaded by only the replica update workload.

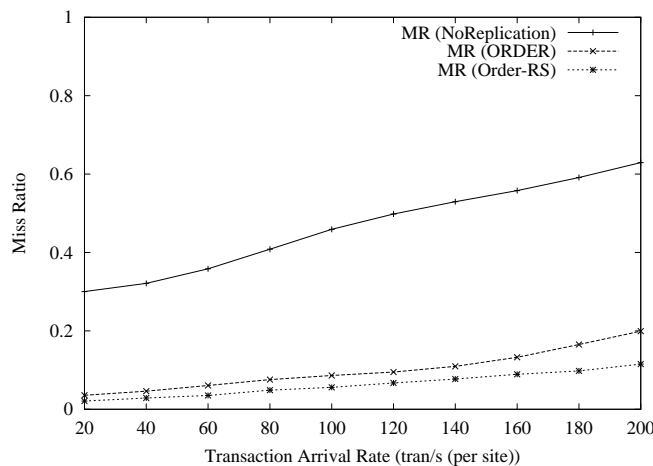


Fig. 8. Performance of ORDER-RS Algorithm

7 Conclusion and Future Work

In this paper, we present two dynamic replication control algorithms designed for medium and large scale distributed real-time database systems. In our algorithms, the periodic transaction declares their data needs and durations. The data replicas are dynamically created based on these data needs. Their update frequencies are also dynamically adjusted with data freshness requirements of the incoming transactions. A detailed simulation study shows that our algorithms

can greatly improve the system performance compared to the systems without replication or systems with simple full replication strategy. It is desirable to implement these replication control algorithms on a real distributed real-time database system and evaluate them with real transaction workloads. We are currently developing a working system prototype. We are also looking into exploiting data similarity and imprecision transaction processing in distributed real-time databases.

References

1. : Naval technology. (In: http://www.naval-technology.com/contractors/data_management/index.html)
2. Wei, Y., Son, S., Stankovic, J., Kang, K.: Qos management in replicated real-time databases. In: 24th IEEE Real-Time Systems Symposium (RTSS 2003). (2003)
3. Son, S.: Replicated data management in distributed database systems. *SIGMOD Record* **17** (1988) 62–69
4. Gray, J., Helland, P., O’Neil, P., Shasha, D.: The dangers of replication and a solution. In: Proc. of the ACM SIGMOD International Conference on Management of Data. Volume 25, 2 of ACM SIGMOD Record., New York, ACM Press (1996) 173–182
5. Wolfson, O., Jajodia, S., Huang, Y.: An adaptive data replication algorithm. *ACM Transactions on Database Systems (TODS)* **22** (1997)
6. Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: Proc. 18th International Conference on Distributed Computing Systems (ICDCS 98), Amsterdam, Netherlands (1998)
7. Wiesmann, M., Pedone, F., Schiper, A., Kemme, B., Alonso, G.: Understanding replication in databases and distributed systems. In: Proc. 20th International Conference on Distributed Computing Systems (ICDCS 2000), Taipei, Taiwan, R.O.C. (2000) 264–274
8. Cook, S., Pachl, J., Pressman, I.: The optimal location of replicas in a network using a read-one-write-all policy. *Distributed Computing* (2002)
9. Yu, P., Wu, K., Lin, K., Son, S.: On real-time databases: concurrency control and scheduling. *Proceedings of the IEEE* **82** (1994) 140–157
10. Son, S.: Supporting timeliness and security in real-time database systems. In: 9th Euromicro Workshop on Real-Time Systems. (1997)
11. Lee, V., Stankovic, J., Son, S.: Intrusion detection in real-time database systems via time signatures. In: 6th IEEE Real-Time Technology and Applications Symposium. (2000) 124–133
12. Shu, L., Stankovic, J., Son, S.: Achieving bounded and predictable recovery using real-time logging. In: Real-Time and Embedded Technology and Applications Symposium. (2002) 286–297
13. Xiong, M., Ramamritham, K., Haritsa, J., Stankovic, J.: MIRROR A state-conscious concurrency control protocol for replicated real-time databases. In: Proc. 5th IEEE Real-Time Technology and Applications Symposium (RTAS 99). (1999) 100–110
14. Andler, S., Hansson, J., Eriksson, J., Mellin, J., Berndtsson, M., Efrting, B.: Deeds: Towards a distributed and active real-time database systems (1996)

15. Mathiason, G., Andler, S.: Virtual full replication: Achieving scalability in distributed real-time main-memory systems. In: Proc. of the Work-in-Progress Session of the 15th Euromicro Conf. on Real-Time Systems. (2003)
16. Peddi, P., DiPippo, L.: A replication strategy for distributed real-time object-oriented databases. In: Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. (2002)
17. Baulier, J., Bohannon, P., Gogate, S., Gupta, C., Haldar, S., Joshi, S., Khivesera, A., Korth, H., McIlroy, P., Miller, J., Narayan, P.P.S., Nemeth, M., Rastogi, R., Seshardi, S., Silberschatz, A., Sudarshan, S., Wilder, M., Wei, C.: DataBlitz storage manager: Main memory database performance for critical applications. ACM SIGMOD Record **28** (1999) 519–520
18. Adelberg, B., Garcia-Molina, H., Kao, B.: Applying update streams in a soft real-time database system. In: ACM SIGMOD. (1995)
19. Ericsson, P.: An operational ship control system in a virtual environment. In: Undersea Defense Technology Europe Conference. (2003)