マルチメディア通信と分散処理 92－1
（1999．2．25）

更新経路の相違に基づく統計的データ・レプリケーション方法

山下 高生　　小野 諭

日本電信電話株式会社

東京都武蔵野市緑町 3-9-11

E-mail: yamasita@slab.ntt.jp　　ono@slab.ntt.jp

本論文では、1) クライアントが要求した最新度以内のデータを提供することおよび 2) クライアントが実際に取得したデータの最新度をクライアントに通知することを実現するデータ複製方法を提案する。本方法は、ある更新が複製データに行われたとき、時間経過に伴って、更新が反映された複製の割合が確率的な意味で次第に増加していくことに着目したものである。最初に、クライアントが要求するとき、およびクライアントに対して実際に提供されたデータの最新度を通知するときに用いる read data freshness (RDF) を定義する。RDF は、クライアントから見たときの確率的な意味でのデータの最新度を表すものである。本方法の update オペレーションは、更新経路を表す木構造上に更新要求を伝搬させることによって行う。read オペレーションは、クライアントから要求された RDF に基づき、この要求を満す複数の複製ノードを選択し、これらの複製ノードから得られたデータのうち、付与されたタイムスタンプが最新のデータをクライアントに提供する。RDF の計算方法としては、実際に測定された更新伝搬の遅延時間の部分的な情報をもとに、複数の複製ノードからデータを取得したときの RDF を、遅延の確率分布に依存しないノンパラメトリック法の区間予測によって行う。

キーワード: データ複製, 弱一貫性, 最新度, 遅延, コーラム・コンセンサス

A Statistical Data Replication Method Based on the Difference of Update Paths

Takao Yamashita , Satoshi Ono

NTT Software Laboratories

9-11, Midori-Cho 3-Chome Musashino-Shi, Tokyo 180 Japan

E-mail: yamasita@slab.ntt.jp　　ono@slab.ntt.jp

In this paper, we propose a data replication method that provides data with at least as much freshness as required by a client and notifies the degree of the freshness of data provided to a client. This method is based on an improvement of the probabilistic rate of updated replicas to all the replicas as time elapses after update starts. We first define a metric, called read data freshness (RDF), that represents the probabilistic freshness of read data from the client viewpoint. Our method uses RDF for specifying the client requirements about freshness and for notifying the degree of the freshness of read data to clients. The read operation of our method is done by acquiring data from replicas and by selecting the latest data according to the associated timestamp. The number of read replicas is changed according to the RDF required by a client. We calculate RDF from measured samples of an update delay using a non-parametric estimation method when a client reads data from multiple replicas. Using a non-parametric estimation method makes our method independent of the probabilistic distribution function of the update transmission delay when update requests propagate among replicas.

Keywords: data replication, weak consistency, freshness, delay, quorum consensus

1 Introduction

With the growth of computer networks, many distributed applications on widely spread computers in a network are sharing various types of data such as those in data stores, telecommunication, decision support, and information retrieval systems. The shared data in a network is kept in a directory service or database systems, and so on. Applications receive services from them using update and read operations. These operations require high availability, scalability, throughput, and a short response time.

In addition, applications require freshness [1] and knowledge of the degree of the freshness if it is insufficient for them. Freshness expresses the idea that an updated value of a data object should be transmitted as soon as possible to applications that read the object. Applications need to know the degree of freshness, because they may change their operation accordingly or users may base their actions on the difference between acquired data and the present estimated data that is computed with freshness. Freshness is a primary requirement for a variety of applications [1] and infrastructures in a distributed computing environment with which they cooperate, such as directory services.

Data replication is an effective technique for attaining high availability, scalability, throughput, and a short response time to operations. It locates replicas in a network that have the same data objects to process operations from clients. There are some ways to maintain values of the same data objects in replicas. These ways change the consistency of replicated data. The consistency of replicated data is classified as strict or weak consistency. Strict consistency includes 1 copy serializability [2]. Examples of weak consistency are epsilon serializability [3]; the consistency by the causal, forced, and immediate operations in lazy replication [4][5]; and the consistency by an asynchronous update function that is implemented in many commercial database management systems. Strict consistency has such a large overhead that it is difficult to improve those properties for update and read operations. On the other hand, data replication with weak consistency enables the properties to be achieved by deferred update at the cost of lower consistency.

Deferred update gradually propagates into replicas after the update operation has been committed at the replica that received it at first. The smaller delay leads to better freshness of data in replicas. However, the increase in the number of replicas worsens the freshness, because updates are transmitted little by little into replicas or are hierarchically propagated in order to reduce the instantaneous load of transmitting an update to many replicas from a replica. Therefore, it is difficult to propagate an update to all the replicas in a time that meets all the requirements of the application. The difficulty of short propagation delay increases the importance of knowing the degree of freshness for some distributed applications.

In this paper, we propose a data replication method that

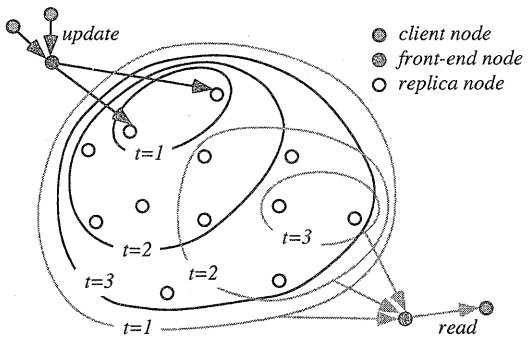

Figure 1: Change in number of updated nodes and nodes that we should access to read the updated data as time elapses.

both provides data with at least as much freshness as required by a client with freshness and notifies the degree of the freshness of data actually provided to a client as a response of a read operation. This method is based on an improvement of the probabilistic rate of updated replicas to all the replicas as time elapses after an update starts. We first define a metric, called *read data freshness* (RDF), that represents the probabilistic freshness of read data from the client viewpoint. Our method uses RDF to specify client requirements about freshness and to notify the degree of the freshness of read data to clients. Second, we propose a way of performing the read operation. The read operation is done by acquiring data from replicas and by selecting the latest data according to the associated timestamp. The number of read replicas is changed according to the RDF required by a client. Third, we propose calculating RDF using a non-parametric estimation method [6] when a client selects the latest data from multiple replicas. This calculates RDF from samples of an update delay measured between replicas whose clocks are synchronized using various time synchronization methods [7][8].

The remainder of this paper is organized as follows. Sec. 2 describes a data replication framework for improving RDF by the cooperation of update and read operations. In section 3 we show how to calculate RDF using a non-parametric estimation method. Sec. 4 mentions remaining problems.

2 Improving freshness by cooperation of update and read operations

There are three types of nodes in our method: client, front-end, and replica nodes. The operations requested by clients are update and read.

First we explain a read operation that provides data with freshness that meets the client requirement and notifies the degree of the freshness of read data. A deferred update modifies data on only some of the replicas when an update operation starts. The number of replicas with the updated data increases as time elapses. Figure 1 shows change in

number of updated replicas. The replica nodes encircled by black solid lines are updated at $t = 1, 2$, and 3, respectively. The increase in number of updated replica nodes with time causes a decrease in the number of replica nodes that we should access in order to read the updated data. These nodes are encircled by gray solid lines in the figure. We can understand this decrease in number of replicas to be read as an increase in the probabilistic intersection between read and write quorums in a probabilistic quorum system [9] with elapsed time.

The formal definition of RDF is time T_p if all the update operations starting more than T_p before somewhere in a network are reflected in read data with probability p. A read operation along with the RDF required by a client is sent to a front-end node when the client requests a read operation. The front-end node sends the read operation to replicas in a set. We denote the set as a read replica set and a replica in the set as a read replica. Replica nodes keep data and a timestamp that shows when the last update operation modifying the data was requested. When read replicas receive the read operation, they return the data along with the timestamp to the front-end node. The front-end node returns the latest data determined using the timestamp to the client. A front-end node manages the relationship between a read replica set and the RDF of the latest data in the set. When a front-end node receives a read operation from a client, it selects a read replica set whose RDF meets the client requirement. When a front-end node returns the latest data, it returns the RDF of the selected read replica set along with the latest data.

Next, we explain the update operation. The number of elements in a read replica set that satisfies the RDF popularly required by clients should be as fewer as possible (ideally one). Therefore in our method we divide replicas into one or more groups so that the maximum delay of update propagation in every group should be less than the popular requirement of clients. When a front-end node receives an update operation from a client, it sends it to one of the replicas in every group. Then the update propagates in every group and finally data objects in all the replicas are updated.

3 Calculating read data freshness

3.1 System architecture

In our method, a front-end node selects a read replica set so that the RDF of the latest data among the replicas in the set meets the client requirement. Since RDF depends on the way an update propagates among replicas in a network, we first describe the system architecture of our data replication method.

Figure 2 shows the system architecture used in our method. As described in the previous section, we divide replicas into one or more groups. Replicas in every group are connected by logical links. The topology of the graph composed of the logical links and the replica nodes in every group is a tree. An update operation sent by a front-end

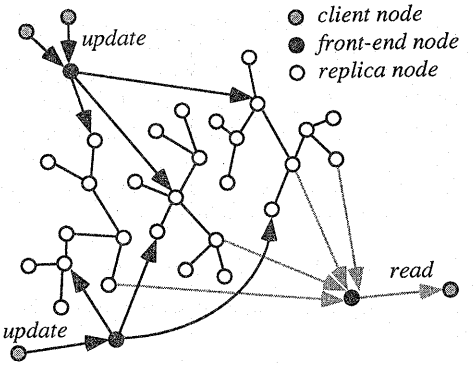

Figure 2: System architecture used in our data replication method

node propagates in the group through the links. We denote a replica node that first receives an update operation from a front-end node as an *update starting replica*. When a replica receives an update operation, it forwards the update to all the adjacent nodes except the replica node from which the update operation came. The update operation finally reaches all the replicas by iteration of the above update forwarding. We denote the minimum subtree whose root is an update starting replica and includes all the replicas in a read replica set as an *update subtree*.

3.2 Overview of freshness calculation

Here, we describe an overview of our RDF calculation method. First, we assign a delay time to a logical link on which an update request propagates. Second, we minimize the update subtree by deleting replicas that do not affect RDF. Third, we calculate RDF when only one update starting replica exists. Fourth, we calculate RDF when multiple update starting replicas exist.

3.3 Assigning a delay time to an update link

When an update request comes to a replica, the request is initially put into the input queue. Next, the replica retrieves the request from the queue, updates the data object, and puts the request into the output queue for transmission to the next replicas. Finally, the request is transmitted to the next replica. Therefore the following delays occur when an update request propagates among replicas.

d_i delay caused by a request waiting for the process to update the data object in an input queue on a replica.

d_p delay caused by carrying out the update operation.

d_o delay caused by a request waiting to be forwarded to the next replica in an output queue on a replica.

d_t delay for transmitting a request from one replica to the next replica.

After an update operation modifies the data object in replica r_p, the request for the operation is transmitted to

its children in the update subtree $r_{c_1}, r_{c_2}, \cdots, r_{c_m}$. Let l_{pc_i} be the link from r_s to r_{c_i}. Let $P_{l_{sc_i}}(t)$ be the probability that the delay on link l_{pc_i} is t. We assign a delay to a link so that $P_{l_{sc_i}}(t)$ and $P_{l_{sc_j}}(t)$ $(i \neq j)$ are independent. For example, we assign delay δ_{sc_i} to link l_{sc_i} in the following two situations.

Situation 1: An update request propagates at the same time from a replica to all its children.

$$\delta_{sc_i} = d_t^{(sc_i)} + d_i^{(c_i)} + d_p^{(c_i)} + d_o^{(c_i)} \quad (1)$$

Situation 2: An update request propagates randomly from a replica.

$$\delta_{sc_i} = d_o^{(s)} + d_t^{(sc_i)} + d_i^{(c_i)} + d_p^{(c_i)} \quad (2)$$

In the above two equations, $d_i^{(n)}$, $d_p^{(n)}$, $d_o^{(n)}$, and $d_t^{(nm)}$ are d_i, d_p, d_o at replica n, and d_t on link l_{nm}. These delays depend on the direction in which the update request is transmitted on a logical link.

3.4 Minimizing update subtree

As an update request propagates among replicas, it brings an update event to a data object at each replica. Times when events occur at replicas have a partial order relation based on the topology of the update subtree used for propagating an update request. We denote a *happened-before* relation [10] as $\prec$. Let e_p and e_c be update events at a replica and its child caused by the same update operation. The relation between e_p and e_c is $e_p \prec e_c$. Let e_i and e_j be update events brought by the same update operation at a replica and one of its descendants. Since the happened-before relation is irreflective and transitive, the relation between e_i and re e_j is $e_i \prec e_j$.

Let $e_i(d_i)$ be the event that is the arrival of an update operation reaching replica i from an update starting replica with delay d_i. Generally, $\mathbb{P}(e_i(d_i) \wedge e_j(d_j))$ is equal to $\mathbb{P}(e_i(d_i)|e_j(d_j))\mathbb{P}(e_i(d_i))$. Let i be an ancestor of j. Since $\mathbb{P}(e_i(d_i)|e_j(d_j))$ is equal to 1 when $d_i < d_j$, $\mathbb{P}(e_i(d_i) \wedge e_j(d_j))$ is equal to $\mathbb{P}(e_i)$. Since $\mathbb{P}(e_i(d_i)|e_j(d_j))$ is equal to 0 when $d_i \geq d_j$, $\mathbb{P}(e_i(d_i) \wedge e_j(d_j))$ is equal to 0. These mean that replica j does not work to improve RDF. Therefore, we calculate RDF in an update subtree that is minimized by excluding a replica whose ancestor is included in the read replica set.

3.5 Estimation of read data freshness using a non-parametric method

An update operation for the same data object can start from multiple replicas. We divide the calculation of RDF into two steps. The first step is to calculate RDF when there is an update starting replica. The second step is to calculate RDF when there are multiple update starting replicas. The calculation of RDF is to estimate the freshness of the latest data in a read replica set. In order to achieve wide applicability of our method, we use an estimation method of confidence interval using a non-parametric method, because a non-parametric estimation method is independent of a probabilistic distribution function. The first step is described in Sections 3.5.1 and 3.5.2. The second step is described in Sections 3.5.3 and 3.5.4. Then sections 3.5.5 and 3.5.6 formally describe the algorithm of the RDF calculation and its communication complexity, respectively.

3.5.1 Estimating a delay on a link

Let X be an old sample of a delay on a link from replica r_p to r_c. Let $X_{(1)}, X_{(2)}, X_{(3)}, \cdots, X_{(n)}$ be order statistics of X. Order statistics are statistics that are rearranged in ascending order [6]. Let $Y_{(1)}, Y_{(2)}, Y_{(3)}, \cdots, Y_{(m)}$ be order statistics of new samples of the delay on the link from replica r_p to r_c. When we rearrange the samples composed of the old and new samples in ascending order, the number of combinations of $Y_{(1)}, Y_{(2)}, \cdots, Y_{(m)}$ is $\binom{n+m}{m}$. All the combinations have the same probability of occuring; it is $\binom{n+m}{m}^{-1}$ [6].

Now we define two functions: *lower delay function* (LDF) and *upper delay function* (UDF). The LDF represents the most lower bound of the probability that a new sample is delay time t based on the order statistics of old samples. The UDF represents the least upper bound of the probability that a new sample is time t based on the order statistics of old samples. Let $\delta_{p,c}^{(l)}$ and $\delta_{p,c}^{(u)}$ be LDF and UDF of the link from r_p to r_c.

$$\delta_{p,c}^{(l)}(t) = \frac{b}{n+1} \quad \text{and} \quad (3)$$

$$\delta_{p,c}^{(u)}(t) = \frac{b+1}{n+1}, \quad (4)$$

respectively where t is the time and b is the number of order statistics $X_{(i)}$ $(1 \leq i \leq n)$ that is equal to or less than t.

3.5.2 Delay from a replica to a read replica set

As a result of minimizing an update subtree in the way described in Section 3.4, the update subtree changes to a tree whose root is an update starting replica and in which, if and only if a node is a leaf of the tree, the node is a replica included in a read replica set.

Let $L_n^{(l)}(t)$ and $L_n^{(u)}(t)$ be LDF and UDF of the delay from replica n to the replicas that are descendants of n and are included in the read replica set. We use $L_n(t)$ to explain facts that are common to both $L_n^{(l)}(t)$ and $L_n^{(u)}(t)$. Let c be a child node of n and $L_{n,c}(t)$ be the delay function (DF) of the delay caused by $L_c(t)$ and $\delta_{n,c}(t)$.

Since we assign the delay to a link so that the delays of the incoming link and outgoing links are independent, $L_{n,i}(t)$ is apparently represented using $\delta_{n,i}$ and L_i as follows, instead of LDF and UDF.

$$L_{n,i}(t) = \int_0^\infty \delta_{n,i}(\tau) L_i(t-\tau) d\tau \quad (5)$$

Strictly speaking, the delays on the incoming and outgoing links described in Section 3.3 are not completely

independent. However, a replica receives update requests for different data objects from multiple points through all the adjacent links. Therefore, we consider that these delays are almost independent due to this mixture of update requests to the replica.

Lemma 1 *$L_n(t)$ is represented using $L_{n,i}$ instead of LDF and UDF as*

$$L_n(t) = \sum_{k=1}^{m_n} \int_0^\infty L_{n,c_k}(t)dt \int_t^\infty L_{n,c_1}(\tau)d\tau \cdots \int_t^\infty L_{n,c_{k-1}}(\tau)d\tau \int_t^\infty L_{n,c_{k+1}}(\tau)d\tau \cdots \int_t^\infty L_{n,c_{m_n}}(\tau)d\tau dt, \quad (6)$$

where m_n is the number of children of replica n and c_k is the k^h child of replica n.

Proof When the delay from n to c_k is minimum and is t, the delay from n to c_i ($i \in \{i : 1 \le i \le n_m\} \setminus \{k\}$) is larger than t. Therefore, the probability that the delay from n to c_k is minimum and is t is represented as

$$L_{n,c_k}(t)dt \int_t^\infty L_{n,c_1}(\tau)d\tau \cdots \int_t^\infty L_{n,c_{k-1}}(\tau)d\tau \int_t^\infty L_{n,c_{k+1}}(\tau)d\tau \cdots \int_t^\infty L_{n,c_{m_n}}(\tau)d\tau dt. \quad (7)$$

Hence, $L_n(t)$ is represented using Eq. (6). □

We calculate $L_n(t)$ by iterating Eqs. (5) and (6) from the leaves to the root of the minimized update subtree. Then we calculate the confidence interval of the minimum value of the delay from an update starting replica n to every replica in the read replica set R using LDF and UDF. Let $P_{n,R}(t)$ be the probability that the minimum value of the delay from n to R is t.

$$L_n^{(l)}(t) \le P_{n,R}(t) \le L_n^{(u)}(t) \quad (8)$$

Therefore, when $L_n^{(l)}(T_p) = p$, the RDF with probability p is T_p.

3.5.3 Separation of common delay function among update starting replicas

Here, we consider the calculation of LDF and UDF when update operations propagate from multiple update starting nodes to replicas in the same read replica set. All the update subtrees have a least common ancestor (lca) of a read replica set. If the lcas of the update subtrees from different update starting replicas are the same, then the subtree from the lca to the read replica set is the same. Therefore, in our method, we reduce the amount of computation by separating the delay function calculation into two parts: calculating the delay from the update starting replica to the lca and calculating the delay from the lca to element nodes in a read replica set.

Lemma 2 *We denote the minimum subtree that includes all the elements in a read replica set as a* read subtree. *Let u be an update starting replica. The lca of all the elements in a read replica set in the update subtree is the nearest replica node to u in the read subtree.*

Proof From the definition of lca, there is one or more pairs of elements in a read replica that satisfy the condition that the lca exists on the path between a pair of replicas. Therefore the lca is in the read subtree. Let a be the nearest replica node in the read replica set to u. Assume that the lca is not a. Let $c_1, c_2, \cdots c_m$ be adjacent replicas of a in the read subtree. Let s_i be the subtree consists of c_i and all the descendants of c_i. This s_i includes one or more replicas in the read replica set. The lca is included in one of s_i ($1 \le i \le m$). Let s_{lca} be the s_i that includes the lca. The path from u to a read replica in s_i ($i \ne lca$) consists of the paths from u to a and the path from a to the read replica in s_i. However, the lca does not exist on the path. Hence, the lca is the nearest replica node to u in the read subtree. □

3.5.4 Delay from multiple replicas to a read replica set

When update operations start at multiple replicas, we calculate the RDF of every update subtree. In our method, we use two types of RDF for multiple update starting replicas. One is the weighted average of all the RDFs and the other is the RDF that has the worst (i.e., biggest) value among all the RDFs. The former aims at applications that need an overall behavior of data update. The latter aims at applications that are sensitive to the worst case of freshness.

3.5.5 Algorithm

Here, we give an overview of the operation of a replica in a read subtree as follows.

Step 1 Construct the minimum read subtree.

Step 2 Find c that is in the minimum read subtree and is the nearest to u.

Step 3 Every leaf replica calculates DF using Eq. (5) and sends it to the adjacent replica.

Step 4 If a replica receives DFs from all the adjacent replicas except s, it calculates DF using Eq. (6) and sends it to s.

Step 5 All the replicas except c terminate this algorithm when the replica sends DF to all the adjacent replicas. Replica c calculates DF using (6) when it receives DF from all the adjacent replicas and then terminates this algorithm.

We describe the above algorithm in a formal way as follows.

R: a set of read nodes.
A: a set of adjacent nodes in the update subtree.
S: a set of adjacent nodes in the minimum read subtree, including read nodes. The initial value is the empty set.

N: a set of nodes to which a message should be transmitted.
nexthop(*node*): this function returns an adjacent node along the path to *node*.

front-end node

$\forall r_k \in R$: $send(\text{"}makeReadSubtree\text{"}, R, R)_{id,r_k}$
(*id*: node id of front-end node)

replica node

States:

$status \in \{out\_of\_subtree, in\_subtree, subtree\_done, calculation\_done\}$,
initial value is $status = out\_of\_subtree$.
$S \subseteq A$
$flagRoot \in \{True, False\}$, initial value is $False$.

Transitions:

$receive(\text{"}makeReadSubtree\text{"}, R, N)_{j,i}$
Precondition: $status = out\_of\_subtree$

Effect:
if $i \in R$ then $N := N \setminus i$
$S := S \cup \{j\}$
$\forall a_k \in A \setminus S$:
$send(\text{"}makeReadSubtree\text{"}, R, N_{a_k})_{i,a_k}$
$(N_{a_k} = \{r | r \in N \wedge nexthop(r) = a_k\})$
$S := S \cup \{a_k\}$
$status := in\_subtree$

become_lca
Precondition: $status = in\_subtree$

Effect:
if $nexthop(u) \notin S$ then $flagRoot := True$
$status := subtree\_done$

start_df_calculation
Precondition: $status = subtree\_done$
$\wedge$ the number of elements of $S = 1$

Effect:
calculate DF P using Eq. (5)
$send(\text{"}DF\text{"}, P)_{i,s}$ $(s \in S)$
$status := calculation\_done$

$receive(\text{"}DF\text{"}, P_j)_{j,i}$
Precondition: $status = subtree\_done$
$\wedge$ the number of elements of $S > 1$

Effect:
$\forall s_k \in S$: if received DFs of $S \setminus \{s_k\}$ then
calculate DF P_k using Eq. (6)
$send(\text{"}DF\text{"}, P_k)_{i,s_k}$
if P_k sent to $\forall s_k \in S$ then
$status = calculation\_done$

3.5.6 Communication complexity of algorithm

Let n_e be the number of logical links in a read subtree. In step 1, since every replica node sends a message for constructing the read subtree, the number of messages in step 1 is n_e. In step 2, every replica node can decide if it is c without sending any messages. In steps 3 and 4, since every replica node sends a message including DF on every incident link, the number of messages in steps 3 and 4 is $2n_e$. Therefore, since the total number of messages used in this algorithm is $3n_e$ for one pair of an update starting replica and a read replica set, the order of communication complexity is $O(n_e)$.

4 Conclusion and remaining problems

In this paper, we proposed a data replication method that both provides data with at least as much freshness as required by a client and notifies the degree of the freshness of data actually provided to a client. We defined a metric, called read data freshness (RDF) for specifying the client requirements about freshness and for notifying the degree of the freshness of data read by clients. Our RDF calculation method has communication complexity $O(n_e)$, when an update operation propagates among replicas that are connected as a tree.

Two problems still remain. One is how to select a read replica set so that its RDF meets the client requirement. The second is how to select an update starting replica so that an update operation can propagate to frequently accessed replicas.

Acknowledgments

We thank Hirohide Mikami for his support and helpful suggestions. We also thank Dr. Haruhisa Ichikawa for his continuous support.

References

[1] E. Pacitti, E. Simon, and R. Melo, "Improving data freshness in lazy master schemes," in *Proc. IEEE 18th International Conference on Distributed Computing Systems*, pp. 164–171, May 1998.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] C. Pu and A. Leff, "Replica control in distributed systems: An asynchronous approach," in *Proc. of the 1991 ACM SIGMOD International Conference on Management of Data*, pp. 377–386, May 1991.

[4] R. Ladin and B. Liskov, "Lazy replication: Exploiting the semantics of distributed services," in *Proc. of the Workshop on Management of Replicated Data*, pp. 31–34, 1990.

[5] R. Ladin, B. Liskov, and S. Ghemawat, "Providing high availability using lazy replication," *ACM Transactions on Computer Systems*, vol. 10, pp. 360–391, November 1992.

[6] S. Shiba and H. Watanabe, *Statistical Method II Estimation*. Shinyosha, 1976.

[7] D. L. Mills, "Precision synchronization of computer network clocks," *Computer Communication Review*, vol. 24, pp. 28–43, Apr. 1994.

[8] T. Yamashita and S. Ono, "A statistical method for time synchronization of computer clocks with precisely frequency-synchronized oscillators," in *Proc. IEEE 18th International Conference on Distributed Computing Systems*, pp. 32–39, 1998.

[9] D. Malkhi, M. Reiter, and R. Wright, "Probabilistic quorum systems," in *Proc. ACM Symposium on Principles of Distributed Computing*, pp. 267–273, 1997.

[10] L. Lamport, "Time, clocks and the ordering of events in distribued systems," *Communication of the ACM*, vol. 21, pp. 558–565, July 1978.