SETIT 2007
4[rth] International Conference: Sciences of Electronic, Technologies of Information and Telecommunications
March 25-29, 2007 – TUNISIA

Improving Distributed Systems Availability using a Pessimistic Server-Based Object Replica Control Mechanism

Mohammad Reza Ebrahimi Dishabi *, Mohsen Sharifi **

* *Islamic Azad University Miyaneh Branch*
mrebrahimy@m-iau.ac.ir

** *Computer Engineering Department*
Iran University of Science and Technology
msharifi@iust.ac.ir

Abstract: Object replication has been used in varieties of distributed systems for the sake of providing increased fault tolerance and availability at the cost of extra added complications and overheads. Part of these overheads is attributable to the mechanisms that are used for the control of object replicas. This paper presents a new mechanism for the pessimistic control of object replicas with comparably lower overheads leading to higher degrees of fault tolerance and availability. It uses the *primary server* approach. A *secondary server* is used too to enhance fault tolerance. It is shown that given p as the probability of server response to a received message, message overhead in the proposed mechanism is $2/p$, and availability is $2p-p^2$, that is quite better than comparable mechanisms and algorithms. It is also shown that algorithms implemented with primary server have better performance than those implemented using voting algorithms.
Key words: Distributed Systems, Reliability, Availability, Fault Tolerance, Object Replication, Synchronization, Primary Server, Voting, Message Overhead

INTRODUCTION

Replication of objects increases system reliability and thus availability required by many applications [3]. Replicating data over a cluster of workstations is a powerful tool to increase performance, and provide fault-tolerance [1,4] for demanding database applications [19,20].Clearly, if multiple identical copies of an object reside on several computers with independent failure modes and rates, then the system would be more reliable. The disadvantage, however, is that the read and write operations performed on a replicated object by various concurrent transactions must be synchronized. This concurrency control problem is known as *replica control problem*[2 ,5 , 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]. Two classes of algorithms have been proposed for replica control in distributed systems: *optimistic* and *pessimistic* [8,7].

Optimistic and pessimistic controls differ in the protocols for delivery of update notifications. Optimistic concurrency control works on the assumption that resource conflicts between multiple clients are unlikely (but not impossible), and allows a process to execute without locking any objects. Only when attempting to commit objects, a check is made to determine if any conflicts have occurred. If a conflict is detected, the process must restart reading the involved object and attempt to make its changes again. If there are few conflicts, there is less redo work, leading to better performance than other concurrency control methods. Unfortunately, if there are many conflicts, the cost of repeatedly restarting processes degrades the system performance significantly and makes the approach unfavorably comparable to other concurrency control methods. Pessimistic schemes, on the other hand, lock objects for the duration they are required by a process. Unless deadlocks occur, a process is assured of successful completion. For instance, two write operations from independent transactions must not be allowed to simultaneously update different copies of the same object. In order to achieve this synchronization between multiple copies, possibly residing at different sites, additional communication and processing cost is incurred [2].

Pessimistic control is implemented by either *primary server* [16 , 8 ,18] or *voting* [16 , 8 ,18] approach. In the primary server approach, one machine is designated as primary. When a replicated

object is to be updated, the change is sent to the primary, which makes the change locally as well as sending a change request to other machines having the replicas. Read can be done from any copy, primary or other replicas. In the voting approach, a client must request and acquire the permission of multiple machines holding object replicas before either reading or writing a replicated object. Most pessimistic algorithms are instances of voting algorithms, known as *quorum consensus* algorithms, in which an operation in the system can proceed only if a group of sites (called a *quorum group*) grants permission. The collection of all possible quorum groups constitutes the quorum set for algorithm. Permission from any one of the groups in the quorum set is sufficient for an operation to proceed.

Two issues have to be considered in the design of any *fault tolerant* pessimistic algorithm. The first issue is *message overhead*. The performance of a replica control algorithm depends on the number of sites involved in the synchronization, as well as the number of messages that have to be exchanged between participating sites. It is important to minimize the message overhead of algorithm [7 ,13].

The second issue that needs to be considered is *availability*. Informally, the availability of an algorithm is the likelihood that a read or write operation initiated by a site will complete in spite of network and site failures. Clearly, in the voting algorithms, a client had to send messages to many machines (*quorum group*). In other words, to get permission for an operation, the client must exchange more then one message between participating sites. However, in primary server algorithms, the client sends only one message to the primary server. Therefore, the primary server algorithms outperform the voting algorithms with respect to message overhead. However, they suffer from possible crash of a single server. Backup servers for primary server may remedy this weakness[7 ,13].

This paper presents a new pessimistic algorithm for object replica control using the primary server approach, with the objective of comparatively better reliability and availability. A secondary server is deployed besides a primary server to enhance the fault tolerance.

The rest of paper is organized as follows. Section 1 discusses the basic principles and assumptions underlying our proposed algorithm. Section 2 presents the design of this algorithm. Section 3 includes an analysis of the availability and message overheads of the algorithm. Section 4 compares the algorithm with other existing algorithms, and Section 5 concludes the paper.

1. Presumptions

We assume a distributed system to be a system consisting of N distinct sites that communicate with primary and secondary machines by sending messages over an unreliable communication network. No assumption is made about the underlying topology of the communication network. However, every site in the system can communicate with any other site when there are no failures. Sites fail only by stopping and site failures are independent. Network links can fail by crashing or by failing to deliver messages.

Given this type of distributed system, four objects located in primary and secondary machines are required to control the system:

1. *Data*: The system creates this original object. Replica objects are copied from this object and broadcasted to other sites.
2. *DataForReplica*: This object saves the name of machines having a copy of the original object (Data). With this object, system knows the hosting places of replicas. The process (program) has to connect to one replica object in order to initiate an operation on that object. Using *DataForReplica*, the system selects a replica object that minimum numbers of processes are bound to it. This is provides a reasonable balance of sending and receiving messages between primary or secondary servers and the clients.
3. *Ibevent*: This object is used to send and receive messages in a distributed system. With this object, clients can send messages to primary or secondary servers in order to get permission for operation on replica objects.
4. StatusOfConnection: This object is used to show the connection status of network. With this object, we can get the name of the machine that has currently permission of an operation on replica object. Figure 1 shows the structure of this object. The *Station* field represents the name of machine that has currently changed permission of an operation on replica object. The *programid* field shows the identity of the program that has tried to change the replicated object, and the *allowedit* field shows the status of connection. The machine can change the object only if the value of *allowedit* is equal to 1.

$$\begin{array}{l} statusofconnection = calss \\ \quad station : string; \\ \quad programid : short\,\mathrm{int}; \\ \quad allowedit : short\,\mathrm{int}; \\ end; \end{array}$$

Figure 1. ***Structure of statusofconnection***

2. Primary Copy with Two Servers Algorithm

Let us call our proposed algorithm *Primary Copy with Two Servers* (PCTS) and describe how it works and how it is implemented.

Consider two machines from a network, one as a

primary server and another as a secondary server. Secondary server is the backup for primary server and replaces it in case of its failure. Primary server starts running when the system starts running. It consequently becomes responsible for keeping the names of machines holding the *DataForReplica* object. It checks for the existence of replica object in these machines, and in case a replica object does not exist in a desired machine, it creates a replica from the original object (i.e. *Data*) and puts it on that machine. Figure 2 shows this behavior.

Suppose, S_m ,..., S_k machines in Figure 2 can host the original copy of object (i.e. *Data*); this information is logged in *Dataforreplica* object. Client programs P_1 ... P_k ... P_n are executing on these machines at *level2*; S_x,P_x denotes the execution of client program P_x on S_x machine. Each running program has a unique identifier system-wide.

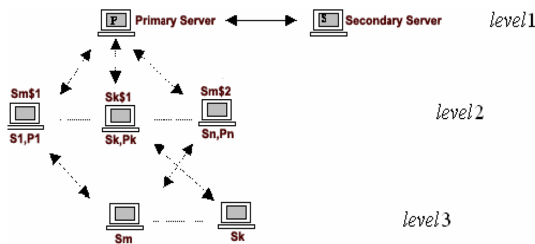

Figure 2. The system behavior

For example, S_m\$1 is the P_1 identifier since it is associated with a replica object located in S_m, and S_m\$2 is the P_n identifier since this is the second program associated with the object located in S_m. If another program requests to connect to the object located in S_m, the system gives it S_m\$3 as identifier. That is to say, whenever the system decides to associate a newly arrived program to a replica object located in a machine named *computername*, it assigns it the *computername$programidas* identifier, where *m* is the maximum identifier number of programs associated with the object, and *programid* is equal to *m*+1.

2.1. Access Permission

When a program wishes to interact with a replica object, it must get permission from the primary server beforehand. Without loss generality, let us consider an example where a program identified as S_m\$1 is executing on S_1 machine and wishes to use the object located in S_m, as shown in Figure 3.

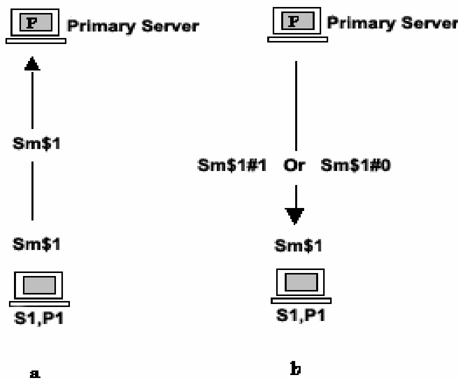

Figure 3. Message Flow for Concurrency Contro

In order to get permission to access an object, the program sends a message to the primary server including its identifier (S_m\$1). Having received the message, the primary server checks the value of *AllowEdit*, which is one of the fields of *Statusofconnection* object. If the value of *AllowEdit* is 1, it sends back S_m\$1#0 message to the program, because another program is currently engaged with the replica object. Otherwise, if the value of *AllowEdit* is 0, the primary server sets *AllowEdit* to 1 and sends S_m\$1#1 message to the program, allowing it access to the object.

2.2. Update Propagation

Updates have to be performed at all replicas . If the update load increases , each replica has less resources to execute queries. Hence,the performance gain from 5 to10 replicas is not as big as from the non-replicated system to 5 replicas. More about this phenomena can be found in [17].

When a program modifies an object, changes must be propagated throughout the system. To achieve this, the program must send a *RefReplicaData* message to the primary server. When the primary server receives this message, it makes the changes locally and sends *RefReplicaData* request message to all machines having the replica.

2.3. Primary Server Crash

If a client does not receive any response to its permission request sent to the primary server after some specified time, let's say 1 minute, it resends its request a number of times, let's say 4. However, if it does not receive any response after repeated this number of tries, it assumes that the server is crashed. This assumption is most probably true, since the server is really downloaded and only works either on permission grants or on doing updates locally.

When a client program recognizes the crash of the primary server, it sends an *ActiveSecondary* message to the secondary server, as is shown in Figure 3. When the secondary server receives the *ActiveSecondary* message, it updates its events locally, in order to respond to messages sent to it, and it sends a SwichToSecondary message to all existing machines in the system; information about which machines exist in the system is obtainable from the *Statusofconnection* object. After receiving the SwichToSecondary message, clients coordinate their activities with the secondary server that becomes responsible for concurrency control in the system thereafter.

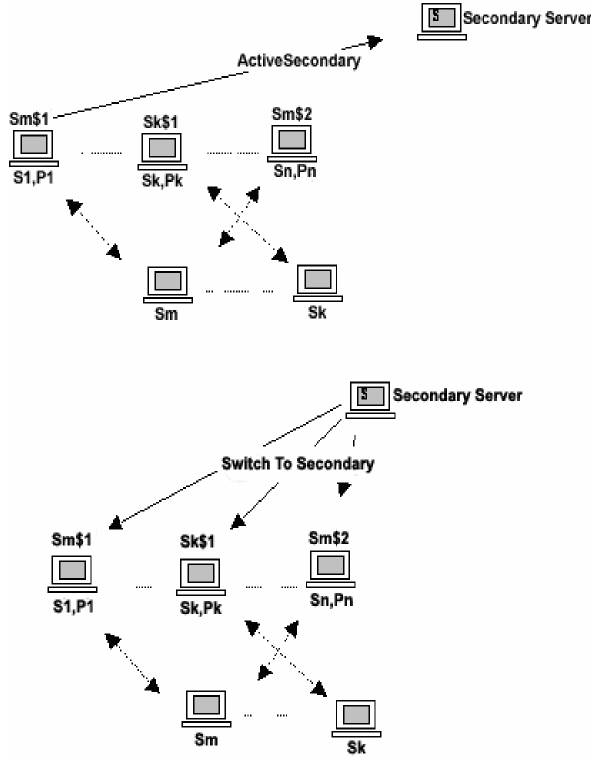

Figure 4. Secondary server becoming the main server

3. System Availability

As mentioned earlier, our system works with two servers, primary and secondary. The secondary server is a backup for the primary server; if the primary server crashes, the secondary server replaces it. Suppose that a client on a machine requests to use a replica object. It must send a permission message to the primary server and assume with probability p that the answer to its request is positive. If the answer is positive, no further messages need to be sent. With probability *1-p* however, the answer can be negative. In this case, more messages must be sent. Therefore, the message overhead of our proposed algorithm in case only one server is used is:

$$M = 1 + (1-p)[1 + (1-p)[1 + (1-p) + ...]]$$

$$\Rightarrow \quad M = \frac{1}{p} \qquad (1)$$

And in case a secondary server is used in addition is :

$$M = \frac{2}{p} \qquad (2)$$

M dentes the number of request message to be sent by the initiating node in order to get permission from primary or secondary server when each server is available with steady-state probability P.

Therefore, the availability of system under our proposition is:

$$A = \binom{2}{1} * p * (1-p) + \binom{2}{2} * p * p = 2p - p^2 \qquad (3)$$

The availability of our system is simply the probability that at least one of servers are functional.

4. Evaluation

Table 1 [10] compares our algorithm with other quorum consensus algorithms. The comparison is with respect to four parameters. We compare four different algorithms with our algorithm. Other than the majority voting scheme and Maekawas's algorithm, we consider two other well-known algorithms.

In HQC [2] or Hierarchical voting consensus, sites are organized in a multilevel hierarchy and voting (with a read and a write quorum) is performed at each level of the hierarchy. The lowest level in the hierarchy contains groups of sites.

In Maekawa's algorithm [7] N sites in a distributed system can be divided into groups with $\sqrt{N}$ sites per group, such that any two groups have a non-null intersection. If S_i and S_j are two groups of $\sqrt{N}$ sites each, then $S_i \cap S_j \neq 0$. There are N such groups, where each group is *associated* with a particular site and each site belongs to $\sqrt{N}$ groups. If a site i initiates an operation, it gets permission from all sites in its associated group S_i.

Another algorithm we consider in our comparison is the RST algorithm [7]. In the RST protocol, nodes are grouped into *N/G* groups of *G* nodes each. Each such group is called a *subgroup.* The *N/G* quorum group is made of $k = \sqrt{N/G}$ subgroups with each subgroup containing G nodes. These quorum groups are constructed in such a way that any pair of quorum groups intersects in exactly one subgroup. This would ensure that read and write operations can be synchronized (no more than one node can obtain a quorum). The construction is such that each node appears the same number of times in exactly $\sqrt{N/G}$ quorum groups, and each such quorum group is associated with G nodes. When a node initiates a read or a write operation, it asks for permission from all the subgroups in its associated group. If permission is granted by all subgroups, the initiating node proceeds with the operation. Permission from a subgroup is obtained if majority of the nodes in that subgroup give their permission. Thus, this is a two level protocol with a requirement of majority at the bottom level inside a subgroup and a quorum at the top level.

The first two compared parameters are the quorum size in the best and the worst cases. In the best and worst cases, our algorithm has the smallest quorum size.

	Majority	HQC	Maekawa	RST	Our Algorithm
1. Quorum Size (best case)	$\frac{N+1}{2}$	$N^{0.63}$	$N^{0.5}$	$\frac{G+1}{2}\sqrt{\frac{N}{G}}$	1
2. Quorum Size (worst case)	$\frac{N+1}{2}$	$N^{0.63}$	$N^{0.5}$	$\frac{G+1}{2}\sqrt{\frac{N}{G}}$	1
3. Message Overhead	$o(N)$	Up to $o(N)$	$o(N^{0.5})$	$o\left(G\sqrt{\frac{N}{G}}\right)$	$o(1)$
4. Is the algorithm fully distributed?	yes	yes	yes	yes	yes
5. Does the algorithm need extra algorithm to get last updates?	yes	yes	yes	yes	no

Table 1. Comparison of our algorithm with related algorithms

The third parameter compares the message overhead for the algorithms. In the majority voting scheme, and the HQC algorithm, the quorum size does not necessarily leads to message overhead. In fact, in the worst case, for these algorithms, a message must be sent to all *N* sites leading to a message overhead of *O(N)*. In our proposed algorithm though, a message have only to be sent to the primary site or the secondary site leading to a message overhead of *O(1)*.

The fourth parameter says that all the algorithms are distributed algorithms where each site has equal responsibility for replica control.

The fifth parameter shows that except our algorithms, all other algorithms need a separate algorithm to get last updates to an object.

We implemented all compared algorithms and ran them on a system consisting of 32 and 64 nodes in a LAN. Figure 5 ,6 , 7 and Figure 8 illustrate the results. Figure 5 and Figure 7 show that our algorithm (*PCTS*) has the lowest average message overhead when the availability of individual nodes are between 0.5 and 0.9 in a 32 and 64-node respectively . Figure 6 shows that if the availability of individual nodes is less then 0.6, the system availability of our algorithm is high. In other words, if the availability of nodes is low, the system availability of our algorithm is better. Figure 8 shows the same result in a 64-node system.

5. Conclusion

In a replica object environment, multiple copies of an object must be kept synchronized by means of a replica control schema. The primary server method is based on requiring a permission from server in order to initiate an operation on a shared object in a distributed system. In this paper, we introduced a new algorithm, using a secondary server based on primary algorithms, and showed that it is possible to reduce the number of messages between nodes to 1 in comparison with other existing algorithms that require higher number of messages. Our (PCTS) algorithm was favorably compared to four related algorithms, namely, majority voting, HQC, Maekawa and RST. Two issues have to be considered in the design of any *fault tolerant* pessimistic algorithm. The first issue is *message overhead.* It is important to minimize the message overhead of algorithm. We showed that PCTS has the lowest average message overhead when the availability of individual nodes are between 0.5 and 0.9 in a 32-node and 64-node system. The second issue that needs to be considered is *availability* and we showed that if the availability of nodes is low, the system availability of our algorithm is better.

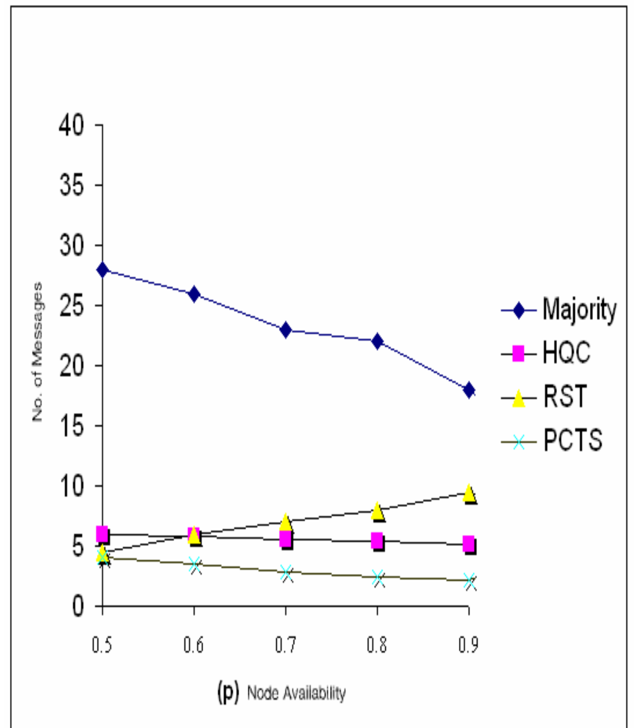

Figure 5. Average message overhead in a 32-node

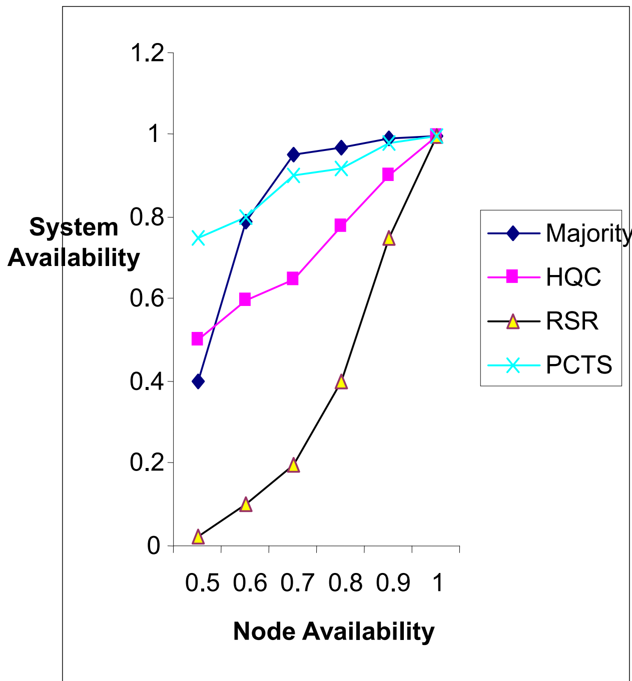

Figure 6. System availability in 32-node

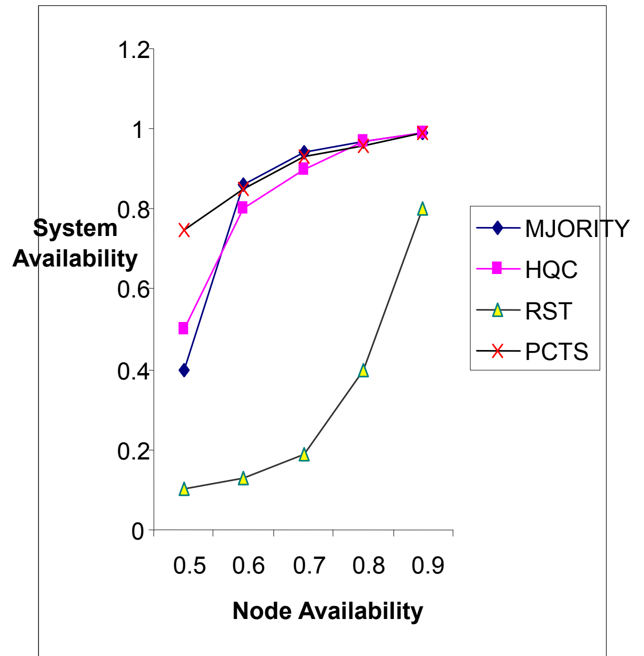

Figure 8. System availability in 64-node

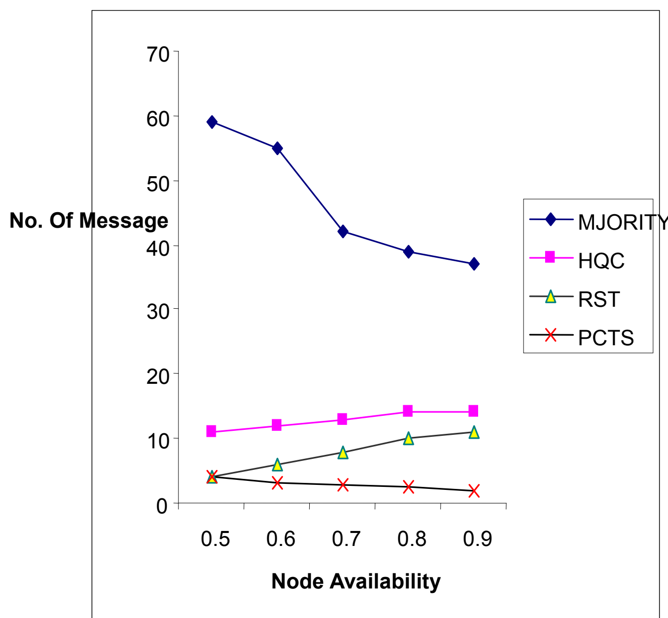

Figure 7. Average message overhead in a 64-node

REFERENCES

[1] V. P. Nelson, "*Fault Tolerant Computing: Fundamental Concepts*", IEEE, 1990.

[2] A. Kumar, "*Hierarchical Quorum Consensus: A New Algorithm for Managing Replicated Data*", IEEE, Sept 1991.

[3] S.K. Shrivastava, G.N. Dixon, G.D. Parrington, "*An Overview of the Arjuna Distributed Programming System*", IEEE, January 1991.

[4] F. Cristian, "*Understanding Fault-Tolerant Distributed Systems*",, IEEE, May 1993.

[5] M.C. Little, D.L. McCue, S.K. Shrivastava, "Maintaning Information about Persistent Replicated Objects in a Distributed System", May 1993.

[6] M.C. Little, S.K. Shrivastava, "Object Replication in Arjuna", BROADCAST Project Technical Report No.50, October 1994.

[7] S. Rangarajan, S. Setia, S.K. Tripathi, "*A Fault Tolerant Algorithm for Replicated Data Management*", IEEE, Dec 1995.

[8] A.S. Tanenbum, "*Distributed Operating Systems*", Prentice-Hall, 1995.

[9] M.C. Little, S.K. Shrivastava, "Using Application Specific Knowledge for Configuring Object Replicas", the 3rd International Conference on Configurable Distributed Systems (ICCDS 1996).

[10] D. Saha, S. Rangarajan, S.K. Tripathi, "*An*

Analysis of Average Message Overheads in Replica Control Protocols", IEEE, 1996.

[11] E. Pacitti, E. Simon, "Update Propagation Strategies to Improve Freshness of Data in Lazy Master Schemes", August 1997.

[12] Anderson, T., Breitbart, Y., Korth, H. and Wool, A., "*Replication, Consistency, and Practicality: Are These Mutually Exclusive?*", *Proceedings of ACM-SIGMOD 1998 International Conference* on Management of Data, Seattle, WA., pp. 484-495, 1998.

[13] R. Strom, G. Banavar, K. Miller, A. Prakash, "*Concurrency Control and View Notification Algorithms for Collaborative Replicated Objects*", IEEE 1998.

[14] M.J. Mahemoff, L.J. Johnston, "*Handling Multiple Domain Objects with Model-View Controller*", IEEE 1999.

[15] K. Hasegawa, H. higaki, M. Takizawa, "Object-Base Protocol for Replicated Objects", IEEE 1999.

[16]PeerDirect.Overview and comparison of data replicationar-chitectures. WhitePaper , Nov.2002.

[17] R. Jim´enez-Peris, M. Pati˜no-Mart´inez, G. Alonso, and B.Kemme,"*Are quorums an alternative for data replication*", ACM Transactions on Database Systems,28(3),2003.

[18] C.Plattnerand G.Alonso. Ganymed: "Scalable replication for transactional web applications" . In Middleware,2004.

[19] Takao YAMASHITA**,** "Dynamic Replica Control Based on Fairly Assigned Variation of Data for Loosely Coupled Distributed Database Systems" , IEICE Transactions on Information and Systems, October 22, 2004

[20] Postgres-(SI): Combining Replica Control with Concurrency Control based on Snapshot Isolation , *"Proc. of the IEEE Int. Conference on Data Engineering, Tokyo",* Japan, 2005