Dynamic Data Recovery for Database Systems Based on Fine Grained Transaction Log

Hong Zhu
Computer School of Sci & Tech
Huazhong University of Sci & Tech
Wuhan, Hubei, P.R. China, 430074
zhuhong@public.wh.hb.cn

Ge Fu*
Computer School of Sci & Tech
Huazhong University of Sci & Tech
Wuhan, Hubei, P.R. China, 430074
fuge@smail.hust.edu.cn

Yi Zhu
Computer School of Sci & Tech
Huazhong University of Sci & Tech
Wuhan, Hubei, P.R. China, 430074
judy@mail.ccnu.edu.cn

Renchao Jin
Computer School of Sci & Tech
Huazhong University of Sci & Tech
Wuhan, Hubei, P.R. China, 430074
renchaojin@gmail.com

Kevin Lü
Brunel University
Uxbridge, UK UB8 3PH
kevin.lu@brunel.ac.uk

Jie Shi
Computer School of Sci & Tech
Huazhong University of Sci & Tech
Wuhan, Hubei, P.R. China, 430074
Shijie1123@gmail.com

ABSTRACT
Data recovery techniques for malicious transactions are increasingly becoming an important issue since the security for DBMSs are mainly prevention based, and they cannot defend systems from unknown attacks. Survivability and availability are essential for modern DBMSs, which require the database provide continuous services in the period of recovery, namely dynamic recovery. In this paper, we presented a data recovery model and introduce extended read-write dependency and phantoms dependency to the model. A fine grained transaction log is proposed for data recovery. The log records all the data items of the read and update-involved operations for the committed transactions, and even extracts data items read by the subqueries in the SQL statements. Based on the log, we develop a dynamic recovery system to implement the data recovery model. The system could provide continuous services while the recovery is processing. Experiments based on TPC-W benchmark show that the dynamic recovery system is high-efficient and reliable.

Categories and Subject Descriptors
H.2.7 [**Database Management**]: Database Administration – *data dictionary/directory, logging and recovery, security, integrity, and protection.*

General Terms
Management, Performance, Design, Reliability, Security.

Keywords
Dynamic recovery, Extended read-write dependency, Phantoms dependency, Fine grained transaction log, Database security.

*The correspondent author

IDEAS08, 2008, September 10-12, Coimbra, Portugal.

1. INTRODUCTION
Survivability and availability are becoming key requirements for modern information systems which require the system provide fault tolerance mechanism: when the damage for data items occurs, the database system should provide continuous, but maybe degraded service while the damage is being repaired [2]. We refer this mechanism as "Dynamic Recovery". There are two evaluation criteria for dynamic recovery: exactness and high-efficiency [1]. Exactness requires that a database system be able to recover from committed malicious transactions. High-efficiency requires that the system should spend as less time as possible on damage assessment and repair. The goal of damage recovery is to locate each affected transaction and recover the database from all malicious or affected transactions [1,2].

Approaches for damage assessment in existing recovery models [1,3,6,7] are based on dependencies between transactions. However, the read operations in subqueries could also generate the read-write dependencies, which are not considered in the existing recovery systems.

Furthermore, there is a special kind of dependency relationship between transactions. For example: two transactions T_1 and T_2 are scheduled sequentially. T_1: *DELETE from item WHERE i_cost > 1000*; T_2: *UPDATE item SET i_cost = 1.2 * i_cost WHERE i_cost > 100*; where T_1 is a malicious transaction. Obviously, T_2 is not dependent upon T_1, and T_2 will be treated as a benign transaction in the previous recovery model [1, 2]. However, if T_1 is undone, the execution results for T_2 are not complete because the values of *i_cost* for those tuples whose *i_cost > 1000* are unchanged. In this case, T_2 should also be treated as affected transactions. Existing recovery models couldn't resolve this kind of dependency relationship.

Existing damage assessment algorithms [1,3,6,7] do not consider the "blind write transaction". A blind write transaction is a transaction whose operations are all update-involved and have no read-write dependency relationships with other transactions. Therefore, blind write transaction would not cause damage

spreading. Moreover, the results produced by benign blind write transactions may turn dirty data items clean and thus prevent damage from spreading. So the results produced by blind write transactions should be kept during data recovery.

2. RELATED WORK

In the area of damage recovery, in [1] the read-write dependency method is first introduced in database systems. An "on-the-fly" recovery method is presented, and the semantics of "dynamic" for data recovery to database system is also proposed. In [3], a model is presented which rewrites transaction history to retain as many as benign transitions in the period of data recovery. In [6], a transaction dependency graph is maintained in the execution period of transactions to substitute the transaction log. A prototype system "Phoenix" based on this model is developed in PostgreSQL DBMS [7]. In [4,9], a prototype system "ODAM" which provides recovery for commercial DBMSs is illustrated. Evaluations for the performance of ODAM are shown in [8]. However, all of these literatures have not solved the issues in Section 1.

3. THE DATA RECOVERY MODEL

Before we propose the recovery model, we first introduce some basic concepts. A transaction history to be repaired is a serializable history generated under the *two phase locking principle*. We denote committed malicious transactions in a history by a set $B=\{B1, B2, ..., Bm\}$; and committed benign transactions in a history by the collection $G=\{G1, G2, ..., Gn\}$. For a transaction history H over $B \bigcup G$, we define $<_H$ as the usual partial order on $B \bigcup G$ for history H, namely, $T_i <_H T_j$ indicates that operations of T_i are scheduled before conflicting operations of T_j. Two operations conflict if they are on the same data item and at least one is write operation. Two transactions conflict if they have conflicting operations [1]. Specially, if transaction T_i and T_j is not conflict transactions, we still use the notation $T_i <_H T_j$ to represent that T_i is scheduled before T_j by the DBMS.

3.1 Transaction Dependency Relationship

3.1.1 Extended Read-Write Dependency

Definition 1. Write Operation. Given a SQL statement *t*, if *t* is a statement for update-involved operation, such as INSERT, UPDATE, and DELETE, then *t* is a Write Operation.

Definition 2. Extended Read Operation. Given a SQL statement *t*, if *t* satisfies one of the following recursive conditions, then *t* is an extended read operation:
1) *t* is a SQL statement SELECT;
2) *t* is a subquery statement of the WHERE sub-clause in a write operation;
3) *t* is a subquery statement in an extended read operation.

Definition 3. Association Degree for a SQL statement. Given a SQL statement *t*, the association degree of *t* is the number of the extended read operations involved in *t*.

Based on the concept of extended read operation, we define the extended read-write dependency relationship between transactions.

Definition 4. Extended Read-Write Dependency. Transaction T_j is dependent upon transaction T_i in a transaction history if there is a data item *x* such that:
1) T_j extended reads *x* after T_i wrote *x* (T_j has an extended read operation on *x* and T_j has a write operation on *x*);
2) T_i doesn't abort before T_j extended reads *x*;
3) every transaction that writes x between the time T_i writes *x* and T_j extended read *x* is aborted before T_j extended reads *x*.

Transaction T_j is extended read-write dependent upon transaction T_i is denoted by $T_j \rightarrow_E T_i$.

3.1.2 Phantoms Dependency

Definition 5. Phantoms Dependency. Given a transaction history *H: ...* , T_1, ... , T_2, ... and *H* satisfies: 1) $\exists$ a write operation *w* in T_1 and an operation *r* in T_2; 2) $T_1 <_H T_2$; 3) $T_2 \rightarrow_E T_1$ does not hold. Suppose that transactions are executed according to another transaction history *H':* ... , T_2, ... (remove transaction T_1 from transaction history *H*). Let s_1 be the collection of tuples operated by *w* in we let *q* reads from the union of the original table and its history table *H* and s_2 be the collection of tuples operated by *r* in *H'*. If $s_1 \bigcap s_2 \neq \varnothing$, then T_2 is phantoms dependent upon T_1.

In the definition above, operations *r* and *w* are phantoms conflict operations. Transaction T_j is phantoms dependent upon transaction T_i is denoted by $T_j \rightarrow_P T_i$.

3.1.3 Transaction Dependency in a Transaction History

Definition 6. Transaction Dependency for a Transaction History. Given a transaction history *H*: $B \bigcup G$, suppose $D=\{<T_i, T_j> | T_i \rightarrow_E T_j$ *or* $T_j \rightarrow_P T_i$, *where* $T_i, T_j \in B \bigcup G\}$; the transaction dependency for *H* is denoted D_H. Then $D_H = t(D)$ (here *t(D)* represents the transitive closure of *D*).

Notation " $\rightarrow$ " is introduced to represent the transaction dependency between two transactions. If $<T_i, T_j> \in D_H$, we denote $T_i \rightarrow T_j$.

3.2 The Recovery Methods

In a transaction history *H:* $B \bigcup G$, where *B* is the set of malicious transactions and $G= \neg B$, we define a set *B* of affected transactions of malicious transactions as follows.

Definition 7. Affected transactions in a Transaction History *H*. Let *A* be the set of transactions affected by malicious transactions set *B*. The recursive definition of *A* is:
1) suppose $<B_k, T_i> \in D_H$, where $B_k \in B$ and D_H is the transaction dependency relationship in *H*, then $T_i \in A$;
2) suppose $T_i \in A$ and $<T_i, T_j> \in D_H$, then $T_i \in A$.

The objective of data recovery is to undo the malicious as well as affected transactions, and retain the execution results of the collection of transactions *G-A* (set of benign transactions). According to the objective, all the data recovery should include:
1) Locate the collection of affected transactions according to the dependency relationship between transactions;
2) Undo the malicious as well as the affected transactions; and
3) Retain the results of other benign transactions except for malicious and affected transactions.

4. FINE GRAINED TRANSACTION LOG

4.1 Definition of Fine Grained Transaction Log

Definition 8. Fine Grained Transaction Log. A Fine Grained Transaction Log is a set of log entries and satisfies the following conditions:
1) a log entry is a six-tuple: [*TRASATIONID, OPTYPE, ITEM, BEFOREIMG, AFTERIMG, TIME*], and each log entry is built according to a data item which a transaction reads or writes;
2) when a transaction is committed, entries for write and extended read operations involved in the transaction are added to the log.

Here, *TRANSATIONID* denotes the transaction that reads or writes the data item. *OPTYPE* is an element in {*SELECT, UPDATE, INSERT, DELETE*}. *ITEM* involves *tablename, rownumber, and column*. For each user table, we modify the table structure and add an automatic incremental field as *rownumber* to identify one tuple of the table. *BEFOREIMG* represents the value before the data item is modified, and *AFTERIMG* denotes the value after the data item is modified. *TIME* is the timestamp when the operation on the data item executes.

4.2 Fine Grained Transaction Log Generator

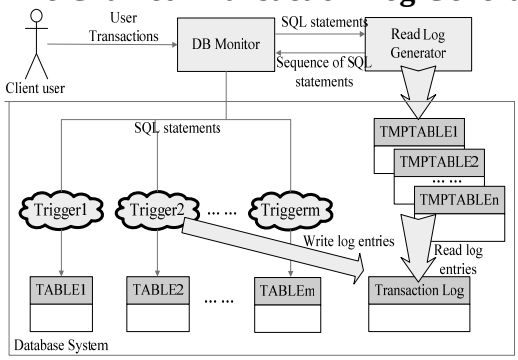

Figure 1. Architecture of Fine Grained Transaction Log Generator

In this section, a fine grained transaction log generator (FGTL generator) is developed to generate the FGTL. The architecture of the FGTL generator is illustrated in Figure 1. There are several components in the FGTL generator: *DB Monitor*, *Read Log Generator*, and some database-involved components (including transaction log table, triggers and temporary tables). Transaction log table records all log entries for the extended read and write operations. The *DB Monitor* is introduced to obtain SQL statements submitted by the *Client user*. The tasks for *DB Monitor* are: 1) extract a SQL statement and submit it to the *Read Log Generator*; 2) after the statement is processed by the *Read Log Generator*, the *DB Monitor* would receive the statements from *Read Log Generator* and send them to the database (a single SQL statement may be divided into several statements); 3) if results are returned from the database, the *DB Monitor* would return the results to *Client user*.

The *Read Log Generator* generates read log entries. Based on the *DB Monitor*, the *Read Log Generator* records data items read by transactions in the temporary tables by a "Divide and Combination" approach. In this approach, a SQL statement is divided into several SELECT ... INTO ... statements to record read data items in temporary tables and then combines the generated statements into a sequence of SQL statements, which has equivalent results with the initial SQL statement (In following Example 1, we illustrate the "Divide and Combination" process). The triggers are used to generate write log entries: for each user table, a trigger is created. If a write operation happens on a data item for a user table, the trigger would write a log entry to the transaction log table. There is a special kind of write operation: "self-modification" operation. For example: *UPDATE item SET i_cost = 1.2 * i_cost*; a self-modification operation implicitly reads and modifies data items based on their original values. For a self-modification operation, the triggers would generate two categories of log entries: the first is the read log entries for the data items on the original values; the other is log entries for real write operations.

Example 1. For the statement: *SELECT ol_qty FROM order_line WHERE ol_i_id IN (SELECT i_id FROM item WHERE i_cost >100)*, the Association Degree is 2. Three SELECT statements are generated for it: 1) *SELECT i_id INTO TMPTABLE1 FROM item WHERE i_cost>100*; 2) *SELECT ol_qty INTO TMPTABLE2 FROM order_line where ol_i_id IN (SELECT i_id from TMPTABLE1*; 3) *SELECT ol_qty from TMPTABLE2*. The three SQL statements are sent to the database sequentially.

5. DYNAMIC RECOVERY BASED ON FINE GRAINED TRANSACTION LOG

As we mentioned in section 4, the transaction log entry is recorded as [TRASATIONID, OPTYPE, ITEM, BEFOREIMG, AFTERIMG, TIME]. To simplify the notation, in the following sections, we use *w[T, x, v1, v2, TIME]* to denote the write operation on data item *x* for transaction *T*, and *r[T, x, v, TIME]* to represent the extended read operation on *x* for transaction *T*.

5.1 Architecture of the Dynamic Recovery System

The architecture of the dynamic recovery system is described in Figure 2. The system includes four parts: *Data Access Confinement Controller (DACC)*, *Damage Assessment Manager (DAM)*, *Transaction Scheduler (TS)* and the *database system.*

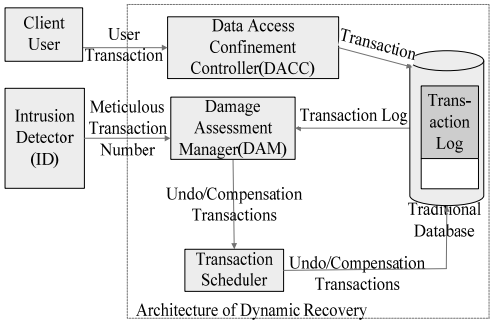

Figure 2. Architecture of the Dynamic Recovery System

The *Intrusion Detector* provides malicious transactions to be repaired. The *DAM* locates affected transactions according to the dependency relationship between transactions. For every malicious or affected transaction, the *DAM* builds an undo

transaction. In order to retain the results produced by a blind write transaction, the *DAM* also creates a compensation transaction which redoes the blind write operations when the records are overwritten by the undo transaction. The *DAM* submits the undo and compensation transactions to the *TS* and then the *TS* sends the transactions to the database system. The *TS* also schedules the undo transaction to prevent conflict while recovery is processing. The *DACC* is applied to prevent damage spreading after a data item is found dirty. In order to confine the damage from spreading, the *DACC* analyzes user transactions submitted by the *Client users* during the recovery period, and would discard the operations which attempt to access dirty data items.

5.2 The Dynamic Recovery Approach

In this section, we discussed important issues in the *Dynamic Recovery System*, including retaining modifications for blind write transactions, the damage assessment and data recovery based on transaction dependency.

5.2.1 Retaining Modifications for Blind Write Transactions

A blind write transaction is a transaction whose operations are all write involved and have no read-write dependency relationships with other transactions. Blind transactions could be easily found in the transaction log: a transaction *T* is a blind write transaction iff ∀ log entry *e* for the transaction *T*, *e.OPTYPE* ≠ *SELECT*. Suppose that *x* is a data item written by a blind write transaction *T*. If *x* is never made dirty or *x* has been cleaned but *x.CLEANTIME* < *w.TIME* (Here each clean item *x* is associated with a number *x.CLEANTIME*, which denotes the time when *x* is cleaned. *w* denotes a log entry of transaction *T* which blind writes *x*), the modifications for blind write operation *w* could be kept in the database. However, in the dynamic recovery, in some conditions, the results for blind write transaction *T* could not be retained, and we denote the condition "Losing Modification Conditions" as follows.

Losing Modification Conditions for blind write transaction *T*

1) ∃ log entry *w[T,x,v1,v2,TIME]* for *T* , *x* is dirty; or

2) ∃ log entry *w[T,x,v1,v2,TIME]* for *T*, *x* has been cleaned and *w.TIME* < *x.CLEANTIME*.

Losing modification conditions for a blind write transaction *T* could be explained that if *x* turns dirty *T* blind writes *x* and turns *x* correctly when *T* executes. At this time, *x* is cleaned by the blind write transaction *T* and would not cause damage spreading any more. However, during the damage assessment, log entries of *T* are skipped by the *DAM*, so that *x* is still treated as a dirty data item, and is recovered to the before image. Therefore, the modifications for the blind write transaction *T* are lost.

In order to retain the results for a blind write transaction which satisfies the losing modification conditions, a compensation transaction is created to redo the blind write transaction. The compensation transaction is just a "copy" of the blind write transaction. After created, the compensation transaction is submitted to *Transaction Scheduler*. When the compensation transaction is executed successfully in the database system, the results modified by blind write transaction could be retained.

5.2.2 Damage Assessment and Data Recovery

The *Damage Assessment Manager* (*DAM*) locates affected transactions according to dependency relationship between transactions. There are two kinds of dependency relationships. The affected transactions caused by extended read-write dependency could be obtained from the transaction log. The *DAM* scans the transaction log entries from the first malicious transaction in malicious transaction collection *B*. The damage assessment process is mainly devoted to three kinds of log entries: 1) the write entry for a transaction in *B*. For this entry, the data item for the log entry is dirty and the correlated transaction should be undone. The *DAM* sends the transaction to the *Transaction Scheduler*; 2) the log entry for a blind write transaction. If losing modification conditions (in Section *5.2.1*) are satisfied, when the last log entry for the blind write transaction is scanned, a compensation transaction is created and sent to the *Transaction Scheduler;* 3) the log entry for a normal transaction. For this kind of log entry, if the transaction reads data items modified by malicious or affected transactions, the transaction is affected and should be undone.

The method to capture affected transactions caused by phantoms dependency is as follows: When the SQL statement *s* which belongs to transaction *T* is executed, an additional SELECT statement *q* following *s*, which almost remains the same appearance with *s*. The difference is that in the "FROM" clause, *q* reads from the union of the original table and its history table (the history table records the version of before images of the latest modifications on the original table, and the data in history table could be obtained from the transaction log). Suppose the collection of the affected rows for statements *s* and *q* are C_1 and C_2 respectively, we check the transaction log to locate the log entries. Those transactions, which have *ITEM.rownumber* field in the collection (C_2-C_1) (The transactions could be obtained from the *TRANSACTIONID* field according to the *ITEM.rownumber* field), are phantoms dependent on *T*. In this way, in the execution time, a phantoms dependency graph for transactions can be established. While the *DAM* scans the transaction log table for data recovery, the phantoms dependency graph is checked. If a transaction T_i *in* the phantoms dependency graph is dependent (either extended read-write dependent or phantoms dependent) on any malicious or affected transactions, T_i is affected, and the *DAM* creates an undo transaction and sends it to the *Transaction Scheduler*.

6. EXPERIMENT RESULTS

6.1 Test Environment

We use SQL Server 2000 as the underlying DBMS. The recovery system, TPC-W platform and the SQL Server database system are deployed in distributed PCs. The database is running on a PC with Windows NT, Pentium R 2.8GHZ CPU, 2GB main memory. The recovery system and TPC-W platform is running on a PC with Windows NT, Pentium R 2.8GHZ CPU, 1GB main memory. These two PCs are connected by a 10/100Mbps switch LAN. The recovery system is implemented using Eclipse on JAVA platform.

6.2 Throughput vs. The Number of Malicious Transactions

The objective of this experiment is to study throughputs under different amount of malicious transactions. In TPC-W platform, the throughput is described using WIPS, WIPSo, and WIPSb. With the amount of malicious transactions increases from 0 to 1,000, we list WIPS, WIPSo, and WIPSb for the dynamic recovery system. NUM_EBS is 1K and NUM_ITEM is 10K.

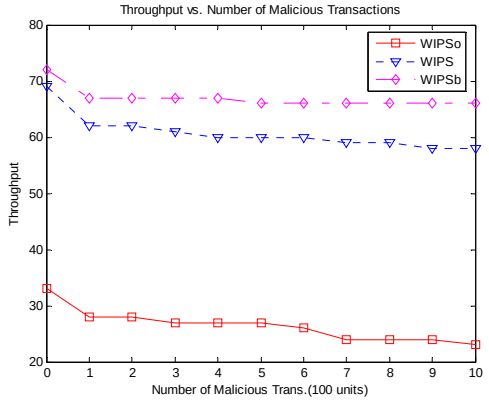

Figure 3. Throughput vs.NUM of MT

Results from Figure 3 show that with number of malicious transactions increases from 100 to 1,000, throughputs decrease by 6.94% to 8.33% for WIPSb, 10.14% to 15.94% for WIPS, and 15.15% to 30.30% for WIPSo. We could conclude that: 1) a large amount of malicious transactions would lead to a low throughput. This can be explained that a large amount of malicious transactions might cause a large number of affected transactions. For each of the malicious and affected transactions *DAM* would generate an undo transaction and the number of undo transactions increased. 2) The throughput decreases the most for WIPSo and the least for WIPSb. This can be explained that only when the write operations are executed, new damages would be generated, and the extended read operations never cause damage spreading. WIPSo measures the throughput with the largest number of write operations (about 50%), so that large amount of damages would be caused and large amount of undo transactions would be created for dynamic recovery system to repair the damages. Therefore, the throughput decreases the largest in WIPSo. In contrast, for WIPSb, most operations (about 95%) are for read, so it is indicated that the damage spreading would not be obvious. Consequently, the number of WIPSb is the highest.

7. CONCLUSIONS

In this paper, we presented a dynamic recovery technique which recoveries a database system from malicious attacks. We introduced the extended read-write dependency and phantoms dependency to the recovery model. Furthermore, a fine grained transaction log (FGTL) which could record data items read by subqueries inside SQL statements is introduced. Based on the FGTL, we implemented the recovery model and developed a Dynamic Recovery System. Experiments based on TPC-W Benchmark show that the throughput decreased by 6.94% to 30.30% for introducing the Dynamic Recovery System. For common applications, the decrease is acceptable. Our future work includes the optimization of the performance using the query optimization, and solves problems on the integrity of the database during the period of recovery.

8. ACKNOWLEDGMENTS

The work presented in this paper is supported by 863 hi-tech research and development program of China, granted number: 2006AA01Z430.

9. REFERENCES

[1] Ammann, P., Jajodia, S., and Liu, P. 2002. Recovery from malicious transactions. IEEE Transactions on Knowledge and Data Engineering. 14, 5 (Sept. 2002), 1167-1185.

[2] Liu, P. 2002. Architectures for Intrusion Tolerant Database Systems. In Proceedings of 18th Annual Computer Security Applications Conference (NW Washington, DC USA, 2002), IEEE Computer Society, 3-13.

[3] Liu, P., Ammann, P., and Jajodia,S. 2004. Rewriting histories: recovering from malicious transactions. Distributed and Parallel Databases. 8,1 (Jan. 2004), 7-40.

[4] Luenam, P., and Liu, P. 2001. ODAM: An On-the-fly Damage Assessment and Repair System for Commercial Database Applications. Technical Report. Dept. of Information Systems, UMBC.

[5] Panda, B. and Giordano, J. In Proceedings of the 1998 ACM symposium on Applied Computing (Atlanta, Georgia, United States, Feb.1998). ACM Press. New York, NY, 253-254.

[6] Pilania, D. and Chiueh, T. 2004. Design, Implementation, and Evaluation of an Intrusion Resilient Database System. In Proceedings of the 20th Annual Computer Security Applications Conference. ACSAC'04 IEEE Computer Society, 179-18.

[7] Rajesh, Y. and Panda, B. 2004. Transaction Fusion: A Model for Data Recovery from Information Attacks. Journal of Intelligent Information Systems Attacks. 23,3 (Nov. 2004), 225-245.

[8] Wang, H. and Liu, P. 2006. Modeling and Evaluating the Survivability of an Intrusion Tolerant Database System. Lecture Notes in Computer Science. Volume 4189 (2006), 207-224.

[9] Yu, M., Liu, P., and Zang, W. 2004. Self Healing Workflow Systems under Attacks, In Proceedings of 24th IEEE International Conference on Distributed Computing Systems (Hachioji, Tokyo, Japan, 24-26 March, 2004). ICDCS'04. IEEE Computer Society, 418-425.