SODA: An Algorithm to Guarantee Correctness of Concurrent Transaction Execution in Mobile P2P Databases

Zhaowen Xing[1] Le Gruenwald[1] K. K. Phang[2]

[1]*School of Computer Science University of Oklahoma, Norman, OK 73019, USA {zhaowenxing, ggruenwald}@ou.edu*

[2]*Faculty of Computer Science and Information Technology, University of Malaya, 50603 Kuala Lumpur, Malaysia kkphang@um.edu.my*

Abstract

This paper proposes an optimistic concurrency control (CC) algorithm, called Sequential Order with Dynamic Adjustment (SODA), that guarantees timely and correct execution of concurrent transaction in Mobile P2P databases. In Mobile P2P, as every peer is fully autonomous, needs timely response and has inherent characteristics (mobility, low bandwidth, limited battery power, limited storage and frequent disconnections), existing CC algorithms cannot be applied directly. SODA is an effort to fill in this gap. The analysis of SODA shows that it is able to improve the response time by applying the concept of sequential order and reduce the transaction abort rate by dynamically adjusting the sequential order.

1. Introduction

Mobile P2P is a collection of mobile peers (e.g., laptops and PDAs) connected via relatively short-range and low bandwidth wireless technologies. Mobile P2P enables the sharing of critical information in applications such as emergency response and homeland security [1]. Since each peer not only provides services but also requests services, it may collect or update its own data, respond to requests and update replica at the same time. To allow these multiple transactions to execute concurrently without violating their correctness, a CC technique is needed.

To the best of our knowledge, no CC technique has been proposed for Mobile P2P databases yet. In addition, the issues of if and how ACID properties can be achieved in static P2P still remain open [2]. In this paper, we design an optimistic CC algorithm for Mobile P2P databases, called SODA. SODA is lightweight and addresses Mobile P2P characteristics. The rest of this paper is organized as follows. Section 2 reviews some related work. Section 3 describes an overview of Mobile P2P and SODA. Sections 4, 5 and 6 present SODA, prove its correctness and analyze its performance evaluation, respectively. Finally Section 7 concludes the paper.

This material is based upon work supported by (while serving at) the National Science Foundation (NSF) and the NSF Grant No. IIS-0312746. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

2. Related work

As cellular mobile networks and Mobile Ad Hoc Networks (MANET) have many characteristics similar to those a of a Mobile P2P network, in this section, we review the CC techniques recently proposed for databases in those networks.

[3] proposed V-Lock, which is based on Summary Schema Model (SSM), for cellular mobile heterogeneous database systems. V-Lock uses global locking tables to guarantee the consistency of global transactions, detect and remove global deadlocks. In V-Lock, local sites use the strict 2PL for serializability. [4] introduced Semantic Serializability Applied to Mobility (SESAMO) for MANET databases. SESAMO is based on semantic serializability and the global serializability is automatically guaranteed if each site maintains its own serializability by applying strict 2PL. Single Lock Manager Approach (SLMA) [5] is proposed for cellular mobile networks, in which only one lock manager resides at a fixed server and handles all lock and unlock requests from the mobile clients. However, these three pessimistic techniques cannot provide timely response due to their blocking nature and frequent disconnections in Mobile P2P.

Partial Global Serialization Graph (PGSG) [6] is introduced for the cellular mobile multi-databases. In

1529-4188/08 $25.00 © 2008 IEEE
DOI 10.1109/DEXA.2008.96

PGSG, to guarantee the correctness of global transactions, a partial global serialization graph is built and checked for cycles; while the local sites serialize transactions using a ticketing mechanism. In [7], mobile clients execute transactions against the local cache, and use strict 2PL to serialize transactions and determine the sequential order. Before transactions commit, the commit request must be validated at the centralized database server by applying one of the three algorithms: Mobile Transaction Commit using Serialization Graph (MTC-SG), SeQuential order (SQ) or SG/SQ. Unfortunately, the drawback of these two optimistic techniques is that both techniques apply serialization graph testing, in which the complexity is always $O(n^2)$ [7].

3. An overview of Mobile P2P and SODA

In Mobile P2P databases, each peer carries its own local database [1], is fully autonomous [2], and shares information in on-the-fly fashion [8]. Therefore, although global transactions (or called remote queries in [8]) do exist, it is unnecessary to maintain the global serializability among peers, which is required in traditional distributed databases and mobile databases [3, 4, 6, 7]. But, every peer must guarantee the correctness of transactions that it processes locally because it may collect or update its own data, reply requests and update replica simultaneously.

In pure (no super-peer) Mobile P2P networks, because no global schema [1, 2] exists, after a peer initiates a transaction T, T is divided into sub-transactions if any, and is broadcasted or flooded as packets to the entire network (or called report pulling [1]). Intermediate nodes, which do not have the requested data, reduce the time to live field by one and forward the packets to the next hop. The process is repeated and eventually, the peers (if exist) containing the required data will receive the packets and process T and flooded the results back to the initiating peer. Unlike in traditional distributed databases, the returned result (called replica in [8]) will be written into the initiating peer's database for future sharing. Therefore, not only the participating peers but also the initiating peers have to run SODA.

If any sub-transaction of T is aborted, then the initiating peer has to abort T as well due to the atomicity requirement. However, if the initiating peer or any participating peer is disconnected in the middle of the execution of T, T will not be aborted immediately. T will be marked as disconnected [6] to allow the peers to reconnect and recover. However, after a predefined timeout, T will be aborted to save the limited system resources.

Inspired by the dynamic adjustment technique proposed in MTC-SG/SQ [7], and based on the combination of Timestamp ordering (TO), OCC [9], and backward validation, we propose an optimistic CC algorithm called SODA.

The design of SODA addresses the issues associated with Mobile P2P. First, as the duration of peer communication is short (due to peer mobility), OCC is applied to avoid unnecessary blocking that exists in pessimistic CC techniques. Second, to address the issue of limited battery power and storage, a sequential order is used to speed-up the validation test and dynamic adjustment is adopted to reduce abort rate. Third, to overcome the problem of frequent disconnections, if a transaction is suspected "disconnected", it will not be aborted immediately.

4. Description of SODA

SODA, given a transaction T, has three phases to go through. *Read and Compute Phase* (Phase 1): T reads the values of a set of data items (called read set, and denoted by RS(T)) and saves them into local variables. When T reads a data item d, a timestamp is assigned (denoted by TS(d)). T also computes the values for a set of data items (called write set, and denoted by WS(T)) and saves them in local variables too. *Validation Phase* (Phase 2): the read set and write set of T are validated against a set of committed transactions. If T passes the validation test, then a timestamp is assigned to T (denoted by TS(T)), and used as the commit time of T and the timestamp of the write set (denoted by WS_TS(T)). WS_TS(T) is $+\infty$ if T is a validating transaction. *Commit and Write Phase* (Phase 3): if T succeeds in the Validation Phase, then it can write the values of the write set into the database and commit; otherwise, T has to be aborted.

Definition 1: Given a validating (or committed) transaction T_1, a committed transaction T_2 and a commonly accessed data item d, T_1 **must-be-serialized-before** T_2 (denoted by $T_1 \rightarrow T_2$) if any one of the following conditions is satisfied:

- *Read-Write (RW) conflict*: $RS(T_1) \cap WS(T_2) \neq \varnothing$ and $T_1 \rightarrow TS(d) < WS\_TS(T_2)$.
- *Write-Write (WW) conflict*: $WS(T_1) \cap WS(T_2) \neq \varnothing$ and $WS\_TS(T_1) < WS\_TS(T_2)$.
- *Write-Read (WR) conflict*: $WS(T_1) \cap RS(T_2) \neq \varnothing$ and $WS\_TS(T_1) < T_2 \rightarrow TS(d)$.

Definition 2: Given a validating (or committed) transaction T_1, a committed transaction T_2 and a commonly accessed data item d, T_1 **must-be-serialized-after** T_2 (denoted by $T_1 \leftarrow T_2$) if any one of the following conditions is satisfied:

- *Read-Write (RW) conflict*: $RS(T_1) \cap WS(T_2) \neq \varnothing$ and $T_1 \rightarrow TS(d) > WS\_TS(T_2)$.
- *Write-Write (WW) conflict*: $WS(T_1) \cap WS(T_2) \neq \varnothing$ and $WS\_TS(T_1) > WS\_TS(T_2)$.
- *Write-Read (WR) conflict*: $WS(T_1) \cap RS(T_2) \neq \varnothing$ and $WS\_TS(T_1) > T_2 \rightarrow TS(d)$.

Assume that T_i's (i=1, …, n) are committed transactions, and T is a validating/committing transaction. If we simply let the validation/commit order be the serialization order like OCC [9], and if there is a RW conflict between T and T_i, i.e. $RS(T) \cap WS(T_i) \neq \varnothing$ and $T \rightarrow TS(d) < WS\_TS(T_i)$ for some data item d, then T is aborted because two orders are different. Such aborts should be avoided if possible. The following sections describe how SODA avoids such unnecessary aborts.

We need a dynamic order among committed transactions other than the validation order. That is, in SODA, a Sequential Order (SO) of committed transactions is maintained as $\{T_1, T_2, \ldots, T_i, \ldots, T_n\}$ (also called a history list, which is ordered from left to right) and can be dynamically adjusted. The dynamic adjustment consists of simple and complex cases. In the simple case, the validating transaction T can be directly inserted into the maintained sequential order without adjustment. And the final sequential order will be: $\{T_1, T_2, \ldots,$ ***low*****, … T,** ***up***$, \ldots, T_n\}$, such that T must-be-serialized-after *low* but before *up*. On the other hand, in the complex case, the sequential order must be adjusted before the insertion of T.

The simple case: Our goal is to find the transactions *low* and *up*, such that SO(*low*) < SO(T) < SO(*up*). If we do, then T passes the validation test (lines 2 to 19 in Fig. 1). $SO(T_i)$ is the function to get the sequential order number of T_i in the history list. For instance, $SO(T_2) = 2$ and $SO(T_i) = i$ if the sequential order is $\{T_1, T_2, \ldots, T_i, \ldots, T_n\}$.

Without loss of generality, we should find two transactions *low* and *up*, where,

$$\boxed{\begin{array}{l} SO(low)=\max\{SO(T_i) \mid T \text{ must-be-serialized-after } T_i,\ 1 \le i \le n\}, \\ SO(up)=\min\{SO(T_i) \mid T \text{ must-be-serialized-before } T_i,\ 1 \le i \le n\}. \end{array}}$$

If *low* (*up*) is not found, then we can conclude that T is not serialized after (before) any other transactions, and we say that SO(*low)* = 0 (SO(*up)*= n + 1) (line 1 in Fig. 1). However, if SO(*low)* = SO(*up)*, then it is impossible for T to be serialized before and after T_i at the same time, thus, T is aborted. If SO(*low)* > SO(*up)*, T should be aborted because it cannot be inserted anywhere in the list. But T passes the validation test if the serialization graph testing is applied instead. Thus, this kind of aborts should be avoided too if possible. The details are given in the complex case below.

The complex case: We have $SO(T_i) < SO(T_j)$ from the maintained sequential order $\{T_1, T_2, T_3, \ldots, \mathbf{T_i}, \ldots, \mathbf{T_j}, \ldots, T_n\}$, but we conclude that $SO(T_i) > SO(T) > SO(T_j)$ after finding *low* and *up*, where *low* = T_j and *up* = T_i. Since T is just stuck between T_i and T_j, if we can find all transactions between T_i and T_j that T must-be-serialized-before directly and indirectly (called T_SB, a double linked list), and if there are no transactions in T_SB that T must-be-serialized-after, then T passes the validation test; otherwise, a cycle is detected and T has to be aborted (lines 20 to 32 in Fig. 1).

```
Boolean SODA(T, History) {
1:  low_index = 0; up_index = History→length() + 1;
2:  counter = 1; // Find transaction up
3:  for (T_i = History→begin(); T_i != History→end(); T_i++) {
4:      if (must-be-serialized-before(T, T_i)) {
5:          up = T_i; up_index = counter;
6:          break;
7:      }
8:      counter++;
9:  }
10: counter = History→length(); // Find transaction low
11: for (T_i =--(History→end());T_i >=History→begin();T_i--) {
12:     if ( must-be-serialized-after(T, T_i)) {
13:         low = T_i; low_index = counter;
14:         break;
15:     }
16:     counter--;
17: }
18: if (low_index < up_index) // The simple case
19:     return true;
20: range = History→subset(up, low); // The complex case
21: T_SB→push_back(T);
22: for (T_i = range→begin(); T_i != range→end(); T_i++) {
23:     for (T_j = T_SB→begin(); T_j != T_SB→end(); T_j++) {
24:         if ( must-be-serialized-before(T_j, T_i)) {
25:             if ( must-be-serialized-after(T, T_i)
26:                 return false; // A cycle is detected
27:             T_SB→push_back(T_i);
28:             break;
29:         }
30:     }
31: }
32: return true; // Got here. T passes the validation test
}
```

Fig. 1: SODA—Validation and Preparation

After T passes the validation test, the sequential order has to be updated to reflect the changes. In the simple case, T is directly inserted in the position just before *up* (lines 1 to 2 in Fig. 2). In the complex case, by looping through all transactions between *up* and *low*, all the transactions in T_SB constructed in the first part of SODA are removed first (lines 3 to 11 in Fig. 2). To construct SO(*low*)<SO(T)<SO(*up*), T and all transactions in T_SB are inserted in the position immediately after *low* (lines 12 to 18 in Fig. 2).

```
void update_SO(low, up, T, History, T_SB) {
    if (SO(low) < SO(up)) // The simple case
        History→insert(up, T);
    range = History→subset(up, low); // The complex case
    T_m = T_SB→begin();
    for (T_i = range→begin(); T_i != range→end(); T_i++) {
        if (T_i == T_m) {
            History→erase(T_i); T_m++;
            if (T_m == T_SB→end())
                break;
        }
    }
    low++; // Insert T immediately after low
    History→insert(low, T);
    // Insert all the transactions in T_SB
    while (!T_SB→empty()) {
        History→insert(low, T_SB→front());
        T_SB→pop_front();
    }
}
```

Fig. 2: SODA—Updating the Sequential Order

Example 1: Let $\{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}$ be a set of committed transactions and the sequential order, and T be a validating transaction at a peer. The read sets, write sets and the timestamps are shown in Table 1.

Table 1: Transaction Information Used in Example 1

	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T
RS	{x}	{y}	{z}	{a}	∅	{b}	{c}	{a}
WS	{z}	{x}	{a}	∅	{b, c}	∅	∅	{b}
TS (d)	5	15	25	35		45	50	28
TS_WS	10	20	30		40			+∞

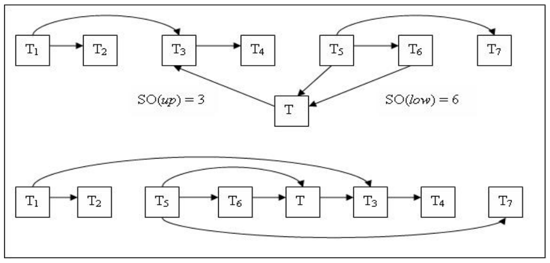

Fig. 3: Validating Transaction T in Example 1

Since $WS(T_3) \cap RS(T) \neq \varnothing$ and $WS\_TS(T_3) > T\rightarrow TS(a)$, T must-be-serialized-before T_3 and *up* = T_3. Similarly, *low* = T_6. Since SO(*low*) > SO(*up*), this is the complex case. T_SB = $\{T_3, T_4\}$, and none of T_3 and T_4 must-be-serialized-before either T_5 or T_6, thus, T passes the validation test. T_3 and T_4 are removed first and then T, T_3 and T_4 are inserted immediately after T_6, the final sequential order is $\{T_1, T_2, T_5, T_6, T, T_3, T_4, T_7\}$ as shown in Fig. 3.

5. Proof of correctness

To prove the correctness (or completeness) of SODA, we must show that any schedule produced by SODA is serializable. To fulfill this goal, we utilize the Serializability Theorem "A schedule S is serializable iff SG(S) is acyclic" [10], that is, we must prove that the new serialization graph is still acyclic after the addition of a newly committed transaction that has passed the validation test.

Lemma 1: Given a sequential order $\{T_1, T_2, T_3, \ldots, T_n\}$ produced by SODA, SODA either does not create any cycles or detects every cycle, if any, in $SG(\{T_1, T_2, T_3, \ldots, T_n\}+\{T\})$ during the validation of any committing transaction T.

Proof: Since the sequential order of transactions complies with their serialization order, every edge (T_i, T_j), if any, must have the same direction. In other words, the edge (T_i, T_j) goes from left to right because $SO(T_i) < SO(T_j)$, where $1 \leq i, j \leq n$.

In the simple case, SODA does not create any cycles in $SG(\{T_1, T_2, T_3, \ldots, T_n\}+\{T\})$: Since *low* and *up* are found and SO(*low*) < SO(T) < SO(*up*), all newly added edges are either (T_i, T) or (T, T_j), where $SO(T_i) \leq$ SO(*low*) and SO(*up*) $\leq SO(T_j)$. Therefore, all existing edges and newly added edges must have the same direction i.e. going from left to right, and thus it is impossible for T to involve any cycle.

In the complex case, SODA captures every cycle in $SG(\{T_1, T_2, T_3, \ldots, T_n\}+\{T\})$: Since *low* and *up* are found, but SO(*low*) ≥ SO(*up*) in the sequential order, and consequently, T may be involved in cycles, such as, T → [*up*] →… T_i … → [*low*] → T, where SO(*up*) < $SO(T_i)$ < SO(*low*), and [*up*] and [*low*] are optional.

Without loss of generality, let the cycle be $T \rightarrow T_{i1} \rightarrow \ldots T_{im} \rightarrow T$, where SO(*up*) $\leq SO(T_{ik}) \leq$ SO(*low*), $i_1 \leq i_k < i_m$, and i_m equals to the number of nodes/transactions in the cycle and between *up* and *low* in the sequential order. Now, we prove that SODA captures every cycle during the validation for T by the induction on i_m.

The basic step, for $i_m = 1$: that is, the cycle is $T \rightarrow T_{i1} \rightarrow T$. Since $T \rightarrow T_{i1}$, the function must-be-serialized-before(T, T_{i1}) returns true (line 24 in Fig.1). Since $T_{i1} \rightarrow T$, the function must-be-serialized-after(T, T_{i1}) returns true (line 25 in Fig.1). Thus, SODA returns false because a cycle is detected (line 26 in Fig.1).

The induction step for $i_m = k$: Suppose every cycle is detected for $i_m \leq k$, that is, the cycle $T \rightarrow T_{i1} \rightarrow \ldots T_{ik-1} \rightarrow T_{ik} \rightarrow T$ is detected because T_SB = $\{T_{i1}, T_{i2}, \ldots, T_{ik-1}\}$, $T_{ik-1} \rightarrow T_{ik}$ and $T_{ik} \rightarrow T$ (lines 22 to 26 in Fig.1). Actually, this cycle is equivalent to T → T_SB → T_{ik} → T. Now, we show that every cycle is detected for $i_m = k+1$. Since $T_{ik-1} \rightarrow T_{ik}$, and T is not serialized after T_{ik} directly, T_{ik} is also added into T_SB (lines 24 to 29 in Fig.1). Since $T_{ik} \rightarrow T_{ik+1}$ and T_{ik} is part of the T_SB and $T_{ik+1} \rightarrow T$, the cycle $T \rightarrow T_{i1} \rightarrow \ldots \rightarrow T_{ik} \rightarrow T_{ik+1} \rightarrow T$ (or T → T_SB → T_{ik+1} → T) is detected as well. Thus, SODA returns false for $i_m = k+1$ (lines 22 to 26 in Fig.1).

Therefore, SODA either captures every cycle, if any, or does not create any cycles in SG($\{T_1, T_2, T_3, \ldots, T_n\}+\{T\}$) during the validation for any committing transaction T.

Theorem 1: If S is a schedule produced by SODA, then S is serializable.

Proof: By Lemma 1, SODA either detects every cycle in SG(S) or does not create any cycles when it validates any committing transaction, so SG(S) is acyclic. Thus, S is serializable according to the Serializability Theorem [10].

6. Performance evaluation

Lemma 2: The time complexity of SODA is $O(p*n^2 + n)$, where n is the number of committed transactions in the sequential order, and p is the probability of a committing transaction conflicting with both *low* and *up* and SO(*low*) ≥ SO(*up*).

Proof: Assume that the number of operations in a transaction is constant and the time to check if two transactions conflict is also constant [7]. *In the simple case (case 1)*: SODA runs one FOR loop after another to find *low* and *up*, and the maximum number of iterations in each loop is *n* (lines 2 to 17 in Fig.1). *In the complex case (case 2)*: SODA runs two nested FOR loops to test the possibility of dynamic adjustment, the maximum number of iterations in each loop is *n*, and the probability of the complex case to happen is *p* (lines 18 to 32 in Fig.1). *In update sequential order* (*case 3)*: SODA runs one FOR loop and one WHILE loop to update the sequential order, and the maximum number of iterations in each loop is *n* (lines 3 to 18 in Fig. 2). By combining the three cases above, the complexity of SODA is $O(p*n^2 + n)$.

For optimistic CC, it has been shown that conflicts among transactions are *rare* [10, 11]. Furthermore, as most transactions in Mobile P2P are read-only, the value of *p* will be very small. Therefore, we can safely say that SODA mostly runs in the linear time. In contrast, the complexity of a serialization graph testing algorithm is always $O(n^2)$ [7]. As stated in Section 4, OCC suffers from unnecessary aborts that SODA has been able to avoid.

7. Conclusion

In this paper, we proposed a CC algorithm, called SODA, which takes into consideration the inherent characteristics of Mobile P2P databases. The processing time of SODA is decreased by applying the concept of sequential order. At the same time, the transactions abort rate is reduced through dynamically adjusting the sequential order.

For future research, SODA will be evaluated with respect to response time and abort rate by means of simulation. An algorithm will be designed to trim the maintained sequential order in order to save the limited storage and to reduce the processing overhead.

References

[1] Y. Luo and O. Wolfson, "Mobile P2P Databases," *Encyclopedia of GIS*, pp. 671-677, 2008.

[2] A. Bonifati, P. Chrysanthis, and et al, "Distributed Databases and Peer-to-Peer Databases: Past and present," Vol. 37, No. 1, pp. 5-11, March 2008

[3] J. Lim and A. Hurson, "Transaction Processing in Mobile Heterogeneous Database System," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 14, No. 6, pp. 1330-1346, 2002.

[4] A. Brayner and F. S. Alencar, "A Semantic-serializability Based Fully-Distributed Concurrency Control Mechanism for Mobile Multi-database Systems," *Proceeding of the 16th International Workshop on Database and Expert Systems Applications*, pp. 1085-1089, 2005.

[5] S. A. Moiz and L. Rajamani, "Single Lock Manager Approach for Achieving Concurrency Control in Mobile Environments," *International Conference on High Performance Computing*, pp. 650 –660, 2007.

[6] R. Dirckze and L. Gruenwald, "A pre-serialization transaction management technique for mobile multi-databases," *ACM Mobile Networks and Applications*, Vol. 5, No. 4, pp. 311-321, 2000.

[7] S. Hwang, "On Optimistic Methods for Mobile Transactions," *Journal of Information Science and Engineering*, Vol. 16, No. 4, pp. 535-554, 2000.

[8] A. Mondal, S. K. Madria, and M. Kitsuregawa, "EcoRepL An Economic Model for Efficient Dynamic Replication in Mobile-P2P Networks," *International Conference on Management of Data*, 2006.

[9] H. Kung and J. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Transactions on Database Systems*, Vol. 6, No. 2, pp. 213-226, 1981.

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

[11] B. Bhargava, "Concurrency Control in Database Systems," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 11, No. 1, January/February 1999.