Conflict Resolution and Reconciliation in Disconnected Databases*

Shirish Hemant Phatak B. R. Badrinath
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
e-mail:{phatak,badri}@cs.rutgers.edu

Abstract

As mobile computing devices become more and more popular, mobile databases have started gaining popularity. An important feature of these database systems is their ability to allow optimistic replication of data by providing disconnected mobile devices the ability to perform local updates. The key problem to this approach is the reconciliation problem, i.e. the problem of serializing potentially conflicting updates from disconnected clients on all replicas of the database. Reconciliation of conflicting updates is a fundamental problem for mobile databases where disconnected updates are allowed. We examine some choices for providing solutions to the reconciliation problem. We also describe a simple but illustrative sample application. Finally we present our conclusions.

1. Introduction

Mobile databases are gaining popularity and are likely to do so well into the future as portable devices become more and more popular and common. One key aspect of these database systems is their ability to deal with disconnection. Disconnection refers to the condition when a mobile system is unable to communicate with some or all of its peers. In such a situation the mobile no longer has access to shared data. Optimistic replication approaches have been proposed to deal with the disconnection problem [29]. In such approaches, the mobile unit is allowed to locally replicate shared data and to operate on this data while it is disconnected. The local updates can be propagated to the rest of the system on reconnection. However, since the local updates potentially conflict with other updates in the system, some form of conflict detection and resolution is required.

*This research work was supported in part by DARPA under contract numbers DAAH04-95-1-0596 and DAAG55-97-1-0322, NSF grant numbers CCR 95-09620, IRIS 95-09816 and Sponsors of WINLAB.

The architectures we consider are extended client server architectures. Here, the primary copies of all data items are stored on the server. All transactions must commit on the server to be "globally" committed. The clients are allowed to locally replicate a subset of the current database state (i.e. the set of all committed versions of all data items currently in the database). Local transactions on the client can operate on this local replica and perform updates. As long as the client is connected to the server, each local transaction is automatically serialized on the server before it is allowed to commit. However, if the client is disconnected and cannot access the server, local transactions are allowed to "locally" commit in the sense that their updates are made available to other local transactions after they have locally committed. The client on reconnection propagates all local transactions on the server for globally serializability testing. A transaction that can't be serialized due to conflicts, or because it read dirty data from another non-serializabile transaction, must be rolled back. Otherwise, the transaction is globally committed and its updates applied to the shared database. We refer to the process of serializability testing of and globally commit of disconnected transactions as reconciliation.

1.1. Related work

Recently lot of research has been directed at optimistic replication schemes and at mobile databases and reconciliation. Some early work can be found in [3, 18, 19]. Recently, Gray et. al. in [15] present a system architecture and a replication model for mobile databases. BAYOU [12], provides another model for mobile data repositories. In BAYOU, information required for the reconciliation process is included in each update to the data repository. Walborn and Chrysanthis [32] provide an application semantics based approach to the reconciliation problem.

Our approach to the reconciliation problem is presented in [28]. In this paper, we present our design choices and decisions, and provide a sample application to show how the algorithm in [28] can be put to work in the real world.

1.2. Organization of the paper

We have organized the remainder of this paper as follows: section 2 describes the reconciliation process and the choices available for reconciliation algorithms; section 3 describes a sample application; and section 4 briefly lists our conclusions.

2. The Reconciliation Process

In most systems the replication (hoarding) and local update phases of a mobile's operation can be dealt with in a straightforward fashion. There are of course notable exceptions, especially when high speed mobility and trickle reintegration are concerned. The reconciliation (reintegration) phase of operation, on the other hand, provides interesting challenges in any disconnected database model. In this section, we explore some of the alternatives we have considered for the reconciliation process.

2.1. Data-centric versus Transaction-centric reconciliation

Reconciliation can occur at various granularities. Two obvious choices are to run the process at the granularity of individual data items or the granularity of entire transactions (i.e. read sets and write sets of transactions). The former is called data-centric; the latter transaction-centric. Data-centric reconciliation is the norm for most commercial disconnected databases.

To understand the difference between the two, let us consider a simple example. Consider a single version database consisting of a single data item, $x = 1$. Further suppose that a mobile client downloads x and disconnects and runs a single transaction $T_{1'} : x = x + 1$. Meanwhile, another transaction T_1 on the server updates $x = 2$. The history here is as follows:

History 1:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1$
Client: downloaded $x = 1$, $r_{1'}[x] = 1, w_{1'}[x = x + 1] = 2, c_{1'}$, reconcile

On reconnecting the reconciler finds that the value of x read by the client transaction $T_{1'}$ i.e. 1 is different from the value of x at the server, 2 (Note that we are assuming a single version system, so that any past values of x are "lost"). Thus a conflict exists and conflict resolution needs to be performed.

In the data-centric approach, the system uses rules to deal with data item conflicts. In our example, the DBA could define the following rule for x-conflicts using the commutativity of addition and subtraction:

$$x^s_{new} = x^s_{old} + x^c_{new} - x^c_{downloaded}$$

Here, x^s_{new} is the new value to be written to the server data set after conflict resolution, x^s_{old} is the value present on the server before conflict resolution, x^c_{new} is the value of x currently present on the client, and $x^c_{downloaded}$ is the value of x that the client downloaded.

Thus, on using this rule on the server, we achieve the following history after conflict resolution:

History 2:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, w_c[x] = 3, c_c$

Here T_c is a new transaction created on the server to reflect the client's updates. Note how the value of x is modified. Also note that transaction $T_{1'}$ has now been transformed into transaction T_c on the server, which is a blind write. This is because $T_{1'}$'s original read set is meaningless and the reconciliation mechanism does not understand transactions. To illustrate this further consider the following history, where two transactions run on the client:

History 3:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1$
Client: downloaded $x = 1$, $r_{1'}[x] = 1, w_{1'}[x = x + 1] = 2, c_{1'}, r_{2'}[x] = 2, w_{2'}[x = x + 1] = 3, c_{1'}$, reconcile

The reconciler merely sees the downloaded image and the after image of x when the client reconciles updates. Thus, the fact that two transactions were run on client is lost (they are both replaced by a single transaction T_c on the server); however we get the following history that is still semantically correct:

History 4:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, w_c[x] = 4, c_c$

The advantage of this approach is that it is fast and cheap to implement. Moreover, the client need not even maintain the transactions it executed, just the current and downloaded values of the data items. However, if the semantics of the client transaction changes, e.g., suppose transaction $T_{1'}$ doubles the value of x, $x = 2x$, then our rule is useless since it might lead to incorrect execution. The problem becomes intractable using the data-centric approach if there are multiple transactions with different semantics. To complicate the issue, very few transactions access only one data item. If some of the updates from a transaction are accepted, while others are rejected, we also compromise the atomicity of the client transactions. The only solution, which is followed by many commercial databases, is to force all updates to the be reconciled failing which the database is taken offline pending manual reconciliation. Obviously, this would lead to loss of throughput and database availability.

To solve these problems, we resort to transaction-centric reconciliation. In this approach, client transactions are reconciled as a unit, one at a time. Thus, entire transactions are reconciled or rejected. The client can also specify the semantics of the transaction in form of conflict resolution routines or functions. Thus, if transaction $T_{1'}$ doubles the value of x, rather than incrementing it our sample history becomes:

History 5:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1$
Client: downloaded $x = 1$, $r_{1'}[x] = 1, w_{1'}[x = 2 * x] = 2, c_{1'}, r_{2'}[x] = 2, w_{2'}[x = x + 1] = 3, c_{1'}$, reconcile

which is identical to the earlier history 1 except for transaction semantics, but leads to a very different history on reconciliation! The history after $T_{1'}$ is reconciled is:

History 6:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, r'_{1'}[x] = 2, w'_{1'}[x = 2 * x] = 4, c'_{1'}$

and after reconciling $T_{2'}$ is:

History 7:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, r'_{1'}[x] = 2, w'_{1'}[x = 2 * x] = 4, c'_{1'}, r'_{2'}[x] = 4, w'_{2'}[x = x + 1] = 5, c'_{2'}$

Note that this is a semantic rather than syntactic redo of the transactions. Also note how the approach easily takes into account the different semantics of transactions $T_{1'}$ and $T_{2'}$. Further, if the semantics of $T_{1'}$ and $T_{2'}$ are unknown, we must reject $T_{1'}$. However, since $T_{2'}$'s read set $\{x = 2\}$ matches the value on the server, we can still reconcile $T_{2'}$, and we get the following history:

History 8:
Server: $w_0[x] = 1, c_0, r_1[x] = 1, w_1[x] = 2, c_1, r'_{2'}[x] = 2, w'_{2'}[x] = 3, c'_{2'}$

Note that only values of x in the read and write sets of $T_{2'}$ are being used and not the semantics of $T_{2'}$. This property is very useful in mobile systems where transactions might need to be prioritized for reconciliation over weak or intermittent links as in case of cellular or infostation (see [13, 14]) coverage. Since transaction dependencies no longer matter, transactions can now be sent to the server in any order. In our example ${T_2}'$ can be reconciled before $T_{1'}$ even though $T_{2'}$ reads dirty data written by $T_{1'}$. Thus, if $T_{2'}$ is "more important" than $T_{1'}$. then it can be sent to the server first.

As this example shows, transaction-centric reconciliation is considerably more powerful than data-centric. However, it should be noted that transaction centric reconciliation implies more work on the part of the client (entire transactions must now be tracked), and more complicated algorithms for reconciliation. In the next subsections we discuss how transaction centric reconciliation can be made even more powerful using weakened serializability models and multiversioning of data items on the server.

2.2. Weakening Serializability Guarantees

The mobile environment is inherently susceptible to weak consistency. This is due to the fact that we allow disconnected client to locally modify their replicas of the database. However, on reconnection, we require that these updates be somehow applied to the server dataset in a serializable fashion. This turns out to be a major hurdle for most systems, since full serializability is a fairly pessimistic and strong guarantee.

It stands to reason, therefore, that weakening the serializability requirement might lead to better performance of the system. Data-centric models inherently rely on this fact to improve performance. This is because the data-centric model assumes a fixed set of semantics for client transactions which is coded into the reconciliation rules. These fixed semantics can be used to bypass the serializability guarantees while still maintaining consistency of the data.

On the other hand, all effort must be made to provide as strong a serializability guarantee as can be achieved with reasonable performance. One such guarantee that works naturally with multiversion systems is snapshot isolation [7]. This guarantee is almost as strong as read committed, but is weak enough to allow us to focus on just the write sets of reconciling client transactions. Snapshot isolation allows histories in which transactions read from a snapshot of the database and where concurrent transactions write distinct data items (i.e. if a data item is shared by two concurrent transactions, only one of the transactions may write it). Note that serializability additionally requires that only one of the concurrent transactions modify shared data items (i.e. if one of the concurrent transactions modify a shared data item, then the other may not write *any* shared data item).

3. A sample application

Our work in [28] embodies much of the suggestions in the previous section. In particular, we have described an algorithm that achieves multiversion transaction-centric reconciliation and provides snapshot isolation to reconciling transactions. Additionally, the algorithm can be easily modified to support full serializability. At the core of our algorithm are two functions defined for each reconciling transaction: the conflict resolution function CR, which attempts to capture transaction semantics and the cost function C which measures the cost of resolving conflicts.

In this section, we describe a sample application. Using this application as a baseline we show that our conflict resolution model is far more powerful than others described in literature since it inherently allows side effects, i.e. the ability for a client to manipulate data that it has not replicated. We achieve this because unlike most other systems, conflict resolution is not optional in our model, rather the conflict resolution function must be invoked every time a transaction is reconciled against a snapshot, even if the snapshot matches the read set. (This requirement can be dropped for efficiency reasons if side effects are not needed and we not wish to detect phantoms.)

Consider a corporate database serving salepersons. The salespersons use laptops to access the centralized database server. Orders are generated or modified at customer sites while the database clients (laptops) are disconnected, and the corresponding transactions reconciled when the database clients (laptops) reconnect.

Let us assume the following schemas (amongst others) exist on the server (primary keys are denoted by a single * while foreign keys are denoted by a double **):

CUSTOMER(CID*,CNAME,ADDRESS,...)
PART(PID*,SUPPLIERID,PNAME)
ORDERBOOK(OID*,PID**,SIZE)
CUSTBOOK(COID*,CID,PID**,SIZE)

Note that these schemas are not meant to be all the relations in the database, rather these are only some of the relations on the database server. A real database can have many more relations.

The **CUSTOMER** relation has the customer information. The **PART** relation contains the supplier information for all parts. The **CUSTBOOK** is the order book for customers, where all the orders for individual customers are logged. This is used to create invoices. **ORDERBOOK** on the other hand represents the orders that need to be sent to the supplier and aggregates all orders for a part from the **CUSTBOOK**. Note that this implies that this set of schemas is not normalized, nevertheless, it does illustrate a typical application.

Let us assume that whenever a salesperson goes out the salesperson locally replicates the **CUSTBOOK**, **PART** and **CUSTOMER** (or a subset of each, since we do not require that entire relations be replicated by the client). **ORDERBOOK** is not replicated, since the client will not generate orders directly to suppliers. However, there is a tight integrity constraint between the **ORDERBOOK** and **CUSTBOOK** whereby all the sum of all orders for a given part in the **CUSTBOOK** must be less than or equal to the ordersize on the **ORDERBOOK**. (In practice, these values might drift as orders go out to the suppliers. However, we shall assume the tight constraint for our example.)

Further, suppose that the salesperson can execute the following types of transactions (amongst others): *neworder* (to book new orders), *modifyorder* (to modify existing orders), and *cancelorder* (to cancel and delete an order). As mentioned before the client can (but does not have to) specify conflict resolution functions. In our example, these functions are integrated into the transaction template, i.e. they are provided as a part of the transaction specification. However, this need not be the case, and the conflict resolution functions can be generated on demand during reconciliation or not at all. The transaction templates for our sample transactions are presented as algorithms 1, 2 and 3. We use standard embedded SQL like statements interspersed with non SQL operations to define the transaction template. (To actually instantiate the template all variables must be replaced by a value.)

Algorithm 1 Transaction neworder

// *get customer info from customer, if not found create a new customer*
select £custid from **CUSTOMER** *where* **CNAME** = *£customername*
if not found **then**
  create new tempid
  insert (tempid,customername,address) into **CUSTOMER** *where* **CID**=*£tempid*
// *now create a new customer order id*
create new temporderid
// *get part information*
select £partid from **PART** *where* **PNAME** = *£partname*
if not found **then**
  // *Couldn't find the part, so abort the transaction*
  abort
// *Enter the order into the local order book*
insert (£temporderid,£custid,£partid,£ordersize) into **ORDERBOOK**

CR:
// *if customer id was temporary, indicating a new customer, get the new permanent id for customer*
if custid is temporary **then**
  select £custid from **CUSTOMER** *with* **CNAME**=*£customername*
  if not found **then**
    // *new customer, get permanent customer id and insert into server*
    create new custid for CUSTOMER
    insert (£custid,£customername, £address) into **CUSTOMER**
// *Now get the permanent order ids*
create new custorderid for **CUSTBOOK**
create new orderid for **ORDERBOOK**
// *now place the order in the server database*
insert (£custorderid,£custid,£partid,£ordersize) into **CUSTBOOK**
// *also update the global order book*
// *Note that the original transaction did not and could not touch ORDERBOOK*
select £partid from **ORDERBOOK**
if found **then**
  update **ORDERBOOK** *where* **PID** = *£partid, set £size = £size + £ordersize)*
else
  create new oid for **ORDERBOOK**
  insert (£oid,£partid,£ordersize) into **ORDERBOOK**

We shall assume for the purpose of this section that the cost function is uniformly 0. However, other cost functions can be easily integrated into the framework provided here.

The CR functions not only capture the actions of the client transactions, but can extend them to capture additional semantics on the server. In our examples, **ORDERBOOK** is used to place orders with part suppliers and does not even exist on the client. However the CR routines always update the **ORDERBOOK**. Since executing the functions is mandatory in our model, we are guaranteed that successful reconciliation implies that these functions will be executed and the global **ORDERBOOK** will always be updated to reflect the correct order sizes. Moreover, notice how the semantics of a cancelorder are different from the semantics of modifyorder or neworder. Such (and more) semantic heterogeneity is impossible to capture in data centric reconciliation. (Note that in data centric reconciliation, we would need to specify one action for all data conflicts on records of the the same relation.) Note that all inserts from the conflict resolution function become the new writeset of the transaction during the reconciliation process.

Algorithm 2 Transaction modifyorder

// get customer order info from customer order book, if not found abort
select £custoid from **CUSTBOOK** *where* **COID**=*£custoid*
if not found **then**
  abort
// Now register the new order
update **CUSTBOOK**, *set* **ORDERSIZE**=*£newordersize*

CR:
// find the order on the server database
select £custoid,£custid,£partid,£ordersize from **COID** *where* **COID**=*£custoid*
// now place the order in the server db
insert (£custorderid,£custid,£partid,£newordersize) into **CUSTBOOK**
// also update the global order book
// Note that the original transaction did not and could not touch ORDERBOOK
select £size from **ORDERBOOK** *where* **PID** = *£partid*
if \$size + \$newordersize - \$ordersize $>$ 0 **then**
  update **ORDERBOOK** *where* **PID**= *£partid, set £size=£size + £newordersize - £ordersize)*
else
  delete from **ORDERBOOK** *where* **PID** = *£partid*

The *neworder* transaction, algorithm 1, is used by the disconnected client to book a new order from a customer. The transaction template we have presented takes into account the fact that the customer record may not be replicated on the client (since the client is free to replicate only a part of the **CUSTOMER** relation). However, note the the non-availability of a part on the client replica of **PART** is treated as a fatal error, since presumably a salesperson must at least replicate information about parts sold by this salesperson. Note that on the client, the neworder transaction only accesses the **CUSTOMER** and **PARTS** relations and manipulates the **CUSTBOOK** relation (since these are the only ones replicated on the clients). Nevertheless, the conflict resolution function does manipulate the global **ORDERBOOK** during the reconciliation process, ensuring that the global state is always consistent. Note that the changes to **ORDERBOOK** made by the conflict resolution function for a given instance of neworder are not applied to the server database until that instance globally commits. Also note that the data that was read and modified need not be explicitly stored by the transaction, since the system automatically stores such values as a part of the transactions read and write sets, which are required by our algorithm.

Transaction *modifyorder*, algorithm 2, is used to change an order from a customer. As specified here it only allows modification of existing customer orders as replicated on the client. This is a strict requirement; the template can be easily modified to allow changes to orders not replicated by the client, and to have the conflict resolution function correctly look up missing orders during reconciliation. This is done by the *cancelorder* transaction.

Algorithm 3 Transaction cancelorder

// delete the order from the order book
// Note that this may fail if the customer's order is not in the order book
delete **CUSTBOOK** *with* **COID** = *£customerorderid*
if not found **then**
  warning: could not find orderid locally

CR:
// delete the order from the ORDERBOOK
// note that this may succeed though the main delete failed if the order
// exists on the server but not on the client's ORDER book
select £custordersize,£partid from **CUSTBOOK** *where* **COID** = *£custoid*
if not found **then**
  error: could not find orderid
else
  select £orderid,£ordersize from **ORDERBOOK** *where* **PID** = *£partid*
  if (\$ordersize $>$ \$custordersize) **then**
    insert (£orderid,£ordersize,£ordersize - £custordersize) into **ORDERBOOK**
  else
    delete £orderid from **ORDERBOOK**
  delete £custorderid from **CUSTBOOK**

Transaction *cancelorder*, algorithm 3, is used to accept cancelations for previous orders. Note that this template is unique in accepting cancelations for orders not replicated on the client database. In such a case the transaction merely records the orderid provided by the customer, and generates a warning. The error is not generated until the conflict resolution function is unable to locate the orderid on the server database.

4. Conclusions

We have presented an algorithm that provides multiversion reconciliation. The algorithm is unique in that conflict resolution and detection are integrated with global serializability testing.

A key concept in our algorithm is that conflict resolution and detection are decoupled from each other. The responsibility for detecting conflicts lies with the server. This is done by performing serializability testing on locally committed transactions from reconciling clients. On the other hand, conflict resolution is the responsibility of the client. The client manages this by providing the conflict resolution and cost functions for each transaction. In the absence of these functions, the server assumes a default, which guarantees snapshot isolation to "unmodified" client transactions. We have illustrated the use of conflict resolution functions using the salesperson example.

References

[1] D. Agrawal and V. Krishnamurthy. Using multiversion data for non-interfering execution of write-only transactions. *Proceeding of the ACM SIGMOD conference*, pages 98–107, 1991.

[2] D. Agrawal and S. Sengupta. Modular synchronization in multiversion databases: Version control and concurrency control. *Proceedings of ACM SIGMOD Conference*, pages 408–417, 1989.

[3] R. Alonso and H. F. Korth. Database system issues in nomadic computing. *Proceedings of the ACM SIGMOD*, pages 388–392, June 1993.

[4] B. R. Badrinath and S. H. Phatak. An architecture for mobile databases. *Department of Computer Science Technical Report DCS-TR-351*.

[5] B. R. Badrinath and S. H. Phatak. Database server organization for handling mobile clients. *Department of Computer Science Technical Report DCS-TR-324*.

[6] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: Beyond commutativity. *ACM Transactions on Database Systems*, 17(1):163–199.

[7] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A critique of ansi sql isolation levels. *Proceedings of ACM SIGMOD Conference*, pages 1–10, 1995.

[8] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[9] S. Ceri and G. Pelagatti. *Distributed Databases—Principles and Systems*. McGraw-Hill, 1984.

[10] P. Chrysanthis. Transaction processing in mobile computing environment. *IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 77–82, Oct. 1993.

[11] S. B. Davidson. Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems*, 9(3):456–481, Sept. 1984.

[12] A. Demers, K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and B. Welch. The bayou architecture: Support for data sharing among mobile users. *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–7, Dec. 1994.

[13] R. H. Frenkiel and T. Imieliński. Infostations: The joy of many-time many-where communications. *WINLAB Technical Report 119*, Apr. 1996.

[14] D. Goodman, J. Borrás, N. B. Mandayam, and R. D. Yates. Infostations: A new system model for data and messaging services. *Proceedings of IEEE VTC*, May 1997.

[15] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. *Proceedings of ACM SIGMOD*, pages 173–182, June 1996.

[16] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan-Kaufmann, 1993.

[17] T. Imieliński and B. R. Badrinath. Mobile wireless computing: Challenges in data management. *Communications of the ACM*, 37(10):18–28, 1994.

[18] R. Katz and S. Weiss. Design transaction management. *Proceedings of the 21st Design Automation Conference*, pages 692–693, 1984.

[19] N. Krishnakumar and R. Jain. Mobile support for sales and inventory applications.

[20] G. Kuenning and G. J. Popek. Automated hoarding for mobile computers. *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Oct. 1997.

[21] G. Kuenning, G. J. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. *Proceedings of the USENIX Summer Conferenc*, pages 291–303, 1994.

[22] P. Kumar and M. Satyanarayanan. Supporting application-specific resolution in an optimistically replicated file system. *Proceedings of the Fourth IEEE Workshop on Workstation Operating Systems*, pages 66–70, Oct. 1993.

[23] H. T. Kung and J. T. Robinson. On optimistic methods of concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[24] Q. Lu and M. Satyanarayanan. Isolation-only transaction for mobile computing. *Operating Systems Review*, 28(2):81–87, May 1981.

[25] P. E. O'Neil. The escrow transactional method. *ACM Transactions on Database Systems*, 11(4):405–430, Dec. 1986.

[26] P. E. O'Neil. *Database—Principles, Programming, and Performance*. Morgan-Kaufmann, 1994.

[27] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall Inc., 1991.

[28] S. H. Phatak and B. R. Badrinath. Multiversion reconciliation for mobile databases. *Proceedings of the 15th International Conference on Data Engineering*, pages 582–589, Mar. 1999.

[29] M. Satyanarayanan. Coda: A highly available file system for a distributed workstation environment. *Proceedings of the Second IEEE Workshop on Workstation Operating Systems*, pages 447–459, Sept. 1989.

[30] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 1997.

[31] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. *Proceedings of the International Conference on Parallel and Distributed Information Systems*, pages 140–149, Sept. 1994.

[32] G. Walborn and P. Chrysanthis. Supporting semantics-based transaction processing in mobile database systems. *Proceedings of the 14th Symposium on Reliable Database Systems*, Sept. 1995.