REPAIRING LOST CONNECTIONS OF MOBILE TRANSACTIONS WITH MINIMAL XML DATA EXCHANGE

STEFAN BÖTTCHER
University of Paderborn, Faculty 5 (EIM) - Computer Science
Fürstenallee 11, 33102 Paderborn, Germany

Abstract: Whenever applications running on mobile clients share XML data within a server-side database, some key requirements are optimized data exchange, transaction synchronization, and the correct treatment of lost connections during application execution. In order to reduce the costs for data exchange, it may be considerably advantageous when the client caches and reuses XML data of previous queries in comparison to delivering the same XML data from server to client repetitively. Furthermore, transactions synchronization has to provide not only the correct treatment of parallel updates, but has to also take into account lost connections. We present a solution for both problems, which combines an exchange of XML difference fragments with an optimized transaction synchronization technique for long transactions that is able to handle lost connections correctly.

Key words: Mobile databases; lost connections; XML; XPath; caching; optimistic synchronization; optimized data transferal.

1. INTRODUCTION

1.1 Problem origin

Whenever mobile clients access XML data which is stored in a server-side database, some of the major problems to be solved are data exchange, transaction synchronization, and lost connections during application.

A standard approach to handle lost connections within client applications that require access to server data, is that the client aborts the running application and restarts it when the connection is re-established. In comparison, our goal is that not all of the work of the client application is lost. Instead the application shall continue after the connection to the server has been re-established, and whenever possible the previous work of the client application shall be saved.

Furthermore, the possible occurrence of lost connections influences the way in which concurrent mobile transactions synchronize their access to a server-side database. Synchronization by 2-phase locking is not appropriate, as a client that loses its connection to the server during transaction execution prevents other client applications from accessing locked data for an unforeseeably long duration.

Finally, both repairing lost connections and correctly synchronizing transactions within mobile information systems rely on data exchange. Whenever small bandwidth connections are a bottle-neck for data exchange, it is preferable to reduce the data transfer required for repairing lost connections or for correct transaction synchronization, to a minimum. This includes reusing old query results still stored in a client's cache wherever possible, and transporting only the difference XML fragment not yet stored but required on the client, from the server to the client.

Our work has been motivated by the development of an XML based information system for e-learning, which uses XPath queries [14] within a client-server environment involving mobile clients. However, we regard the application field to be much broader, i.e. we regard our technique to be useful wherever XML database data has to be shared by applications running on mobile clients. Figure 1 shows our overall system architecture.

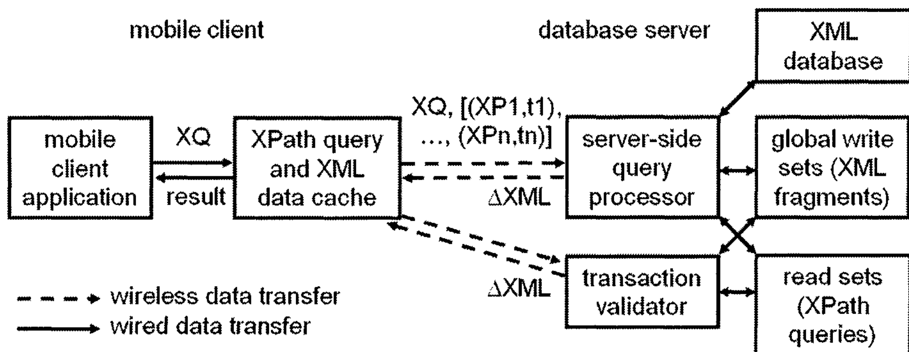

Figure 1. System Overview

1.2 System overview

As shown in Figure 1, a mobile client application does not access an XML database directly. Instead it communicates via a client cache for XPath queries and XML data, which has a wireless connection to a server-side query processor that accesses the XML database. The communication via a client cache and server-side query processor will be used to reduce data exchange and to repair lost connections in such a way that transaction synchronization is still correct. For the purpose of transaction synchronization, the mobile client's cache communicates with a server-side transaction validator. This validator synchronizes the mobile client's transaction – independent of whether or not it has been interrupted by a lost (and re-established) connection.

As low bandwidth is a key problem (at least for our application), we focus on sending small ΔXML fragments (also called XML difference fragments) from the server to the client for all three purposes: ordinary query processing, transaction synchronization, and data status checks after re-establishing a lost connection. In order to support the exchange of XML difference fragments, the client has to inform the server about which data is still stored in the client's cache. Within our system, the client informs the server about the results of previous queries (XP1, …, XPn), the results of which are still in the client's cache. Furthermore, for each query result, the client stores a timestamp that indicates the last time when the query result was refreshed by a difference fragment from the server.

The server uses the previous queries and their timestamps to compute the actual difference XML fragment. Only this difference XML fragment has to be submitted to the mobile client. Thereafter, the client integrates the difference XML fragment with its previous query results. Finally the client uses its refreshed cache to answer the query. (In some cases it may even be possible that a client answers a query without any access to the server, based on the data of its cache alone. We outline this further in section 6.3 including the consequences for validation).

The same technique, i.e. submitting a difference XML fragment from the server to the client, is used at the end of each transaction. This difference XML fragment contains new values for outdated data that have been used within the client's transaction. If this difference XML fragment is empty, the client then knows that its transaction has been committed by the server. Otherwise, the client already has the new values of the outdated data and can restart the transaction using this fresh data.

1.3 Related work, focus of this contribution and problem definition

Our system as given above has to solve a variety of practical problems, including data and query caching, query processing, transaction synchronization and cache consistency checking, and handling lost connections. Within this paper, we present an optimistic transaction synchronization protocol, and we focus on the reduction of data exchange between server and client and on the treatment of lost connections.

While some contributions to mobile transactions relax or redefine transaction properties (e.g. [7]), we follow the argumentation of [6] and support the classical ACID properties. Like other approaches in mobile transactions (e.g. [4]), we combine validation with client side data caching, however we do not update the clients' caches by a server initiated broadcast, but leave the decision to refresh the read data up to the client. According to [1] and [12], this outperforms all other approaches to cache consistency in client/server architectures like ours, where application processing is performed at the mobile client.

Like other approaches to client-side data caching, we let the client reuse previous query results which are still in the client's cache. The server transfers only XML difference fragments to the client, i.e. that data which is needed but missing in the client's cache. We do not discuss how to compute the needed difference fragments and how to integrate these different fragments with the previous query results, as this is already described in [2]. Instead we focus on the data exchange required for transaction processing and transaction synchronization, and on the reuse of cached results in case of lost connections and transaction restarts.

Another problem excluded from this paper is how the client can check, without access to the server-side XML database, that a new XPath query XQ can be answered by using a cached previous query result described by an XPath expression XP. This is the case when XQ selects a subset of XP, independently of the state of the XML database, which can be proven by an XPath containment test. There are a variety of solutions proposed for XPath containment tests, e.g. [5,10,11], whereas our system uses an implementation based on query graphs [3].

One major problem solved in this paper is whether and how the client can continue a transaction which has been interrupted by losing the connection to the server. Another problem also discussed in this paper, is when and how cached query results of previous transactions can be reused. A related problem is to reduce the necessary data exchange from server to client when a transaction has to be restarted because of a synchronization conflict.

Finally, we investigate how to reduce the data transfer needed for synchronization purposes from client to server.

The remainder of this paper is organized as follows. Section 2 summarizes how difference fragments are computed at the server's side and how they are integrated with previous query results within the client's cache. Section 3 describes how the client reads and writes data and what are the options of the client with respect to the use of old data. Sections 4 and 5 focus on status check and synchronization issues. Section 6 describes how to treat transactions when repairing lost connections and outlines opportunities for cross transaction optimization. Finally, Section 7 contains the summary and conclusions.

2. THE USE OF LOCATION INFORMATION FOR FRAGMENT INTEGRATION

In order to reduce the amount of data exchanged between client and server wherever possible, validation information, status check information, and answers to queries XQ are sent in the form of XML difference fragments from the server to the client. As XML difference fragments contain the XML data required but not yet stored in the client's cache, the server has to compute the XML different fragment, and the client has to merge it with previous query results stored in the client's cache.

Together with a query XQ, the client informs the server about the results of previous queries XP1, ..., XPn which are still stored in the client's cache. Given all these query expressions, the server computes the difference fragment that contains the answer to XQ but not the answer to XP1, ..., XPn.5

When the client shall merge an XML difference fragment with a stored fragment of previously retrieved data, it is essential for the client to know the position where the new fragment has to be inserted. This includes that the client has to know under which parent node and after which sibling node a fragment has to be inserted into the client's cache, such that parent-child relationship and sibling order in the client's cache reflect the relationships found in the server's XML database.

In order to solve this problem, we use a node numbering schema which is an extended version of the node number scheme presented in [2]. Our node

[5] Note that client and server can use and communicate query IDs instead of XPath expressions for describing previous query results. This would further reduce the size of data exchanged by the client and the server, but it would require the server to store the XPath expressions and the associated IDs for all previous queries.

numbering scheme assigns a sibling sequence number between 0 and 1 to each node and respects the sibling-order as follows. Whenever C1, ..., Cn are all the child nodes of the same node P and n>=2, then the sibling sequence number of Ci is less than the sibling sequence number of Ci+1 whenever Ci is a preceding sibling of Ci+1. Whenever a fragment is transferred from client to server, each node in this fragment is augmented with the sibling sequence number of the node. Furthermore, each fragment is augmented with the list of sibling sequence numbers of all its ancestors up to the root, called the root path of this fragment. This ancestor path to the root, containing all the sibling sequence numbers from the root to the node or fragment of interest, can be used by the client to find the correct position in its partial copy of the server's XML document. The same sibling sequence numbers are also used within the fragment stored in the client's query cache, such that query results retrieved from the server can be easily merged with the client's fragment.

3. QUERY PROCESSING AND WRITE OPERATIONS AT THE CLIENT'S SIDE

In order to reduce client-server communication to an absolute minimum, the client works as long as possible with its own cache (with the exception of status check operations which are described below). This affects the way in which read and write operations are performed on the client as follows.

3.1 The client's local write set

The client's write operations are stored in a local write set in the client's cache. This local write set is transferred to the server at the end of the transaction. This reduces communication steps from client to server when the same XML fragment is changed multiple times within a client transaction (this is quite typical for experimental applications where data values have to be adjusted).

The client's local write set stores inserted and updated XML fragments together with their root paths which uniquely determine their position within the complete XML document. Furthermore, the client's local write set stores root paths to the deleted fragments of data for each of the client's delete operations. However, for the delete operation it is not necessary to transfer deleted fragments back to the server, i.e. the root paths to deleted fragments are sufficient in order to identify and locate fragments to be deleted from the XML database.

3.2 The client's options for query processing

Whenever a read or write operation is transmitted from the client application to the client's cache, the client's cache computes an XPath expression XQ that describes a sufficiently large fragment which contains all the data to be read or written, as described in [9]. For each query XQ, the client has the choice to use the following query processing algorithm (which is the default case), or to use the status check operation outlined below (which may be used after lost connections or for very old cached data).

3.3 The query processing algorithm of the client's cache

The client's query processing algorithm (Algorithm 1 outlined below) shows the procedure Client.query(XQ) that is called on the client cache for each query XQ that the client application submits. At first, the client collects the XPath expressions XP1, ..., XPn of previous queries, the results of which are still in the client's cache (line (2)). Then the client uses a containment test [3] in order to check whether or not the result of the query XQ can be computed from cached results of previous queries XP1, ..., XPn alone (line (3)). If this is the case, the query XQ is answered locally and no server interaction is needed (line (4)). Only if this is not the case (lines (5)-(11)), the query XQ is submitted to the server together with the list of previous queries, the results of which are still available in the client's cache. The server computes and returns an XML difference fragment of the data needed for answering XQ but not yet contained in the client's cache (lines (6)-(8)) by using a method described in [2]. Furthermore, the server determines which of the previous query results are needed to answer the current query XQ, such that the client can displace other query results whenever it needs memory space for the difference fragment received from the server. The difference XML fragment is merged into the client's cache as described in Section 2, such that the client cache contains a copy of the server's fragment for XQ (line(9)). Thereafter, the timestamp for XQ received from the server is stored in the client's cache (line (10)) for the purpose of transaction validation or status checks after re-establishing a lost connection. Finally (line (11)), the client's copy of the fragment for XQ is merged with the client's own changes (which are stored in the local write set). The resultant data is used within the client's application.

```
(1)   Client.query( XQ )
(2)   { collectPreviousQueries( XP1, ..., XPn ) ;
(3)   if ( XQ can be computed from results of previous queries
      XP1,...,XPn)
(4)   return localResultFor( XQ ) ;    // XQ can be answered locally
(5)   else // get the difference fragment required to answer XQ
(6)   {XMLnewFragment = getDifferenceFromServer(XQ,(XP1,...,XPn)
      ) ;
(7)   // loading the difference from server
(8)   // includes the replacement of fragments if necessary.
(9)   integrate( XMLnewFragment ) ;
(10)  XQ.setTimeStamp = XMLnewFragment.getTimeStamp();
(11)  XQ.applyLocalWriteSet() ;
(12)  }}
```

Algorithm 1: The client's main query processing algorithm

3.4 The client's commit request operation

The client informs the server about the intended completion of a transaction by submitting a commit request to the server at the end of each transaction. Together with the commit request operation, the client transfers its local write set to the server. Furthermore, the client informs the server about (further) previous query results that have been read from the client's cache as part of the actual transaction – together with the timestamp for each previous query result. Because we allow the client to reuse old query results even after lost connections and also from previous transactions, the server has to know which query results the client transaction relies upon, and at what time these query results have been retrieved.

The server answers the client's commit request with an XML difference fragment which returns new values of outdated data in the client's cache that the client has sent as part of its commit request. The XML difference fragment also identifies paths to deleted fragments. If this XML difference fragment is empty, the client then knows that the server has successfully validated the client's transaction. Otherwise, the client knows that its commit request has failed, and the client can merge the new values with its cache and restart the aborted transaction. In both cases, the client takes the timestamp returned with the difference XML fragment as the actual timestamp of the queries which were sent to the server as parameters of the commit request.

3.5 The client's status check request operation

As within the commit request operation, within the status check request operation the client informs the server about previous query results that have been read from the client's cache or are still in the client's cache – together with the timestamp for each previous query result. Again, the server answers the client's status check request with an XML difference fragment which returns new values of outdated data in the client's cache and which identifies paths to deleted fragments. If this XML difference fragment is empty, the client then knows that the previous query results are still up to date. Otherwise, the client knows which previous query results are not up to date. If the client has used this outdated data within the current transaction, the client knows that it has to abort the transaction. If however the XML different fragment contains only data which the client intended to use within the current transaction, the client can simply apply the difference fragment to its cache in order to replace the outdated data. Because the client has refreshed all the previous query results which have been checked, the client can continue with the current transaction.

Transactions can use the status check operation at any time in order to check the status of the cached data. This may be especially useful at the beginning of a transaction when the data to be read is known in advance and the previous query result containing this data has a rather old timestamp.

Again, the timestamp is actualized for every XPath query expression that the client used in the status check request operation.

4. THE SERVER'S IMPLEMENTATION OF THE STATUS CHECK

4.1 The server-side data structures: global write set and transaction's read set

The server's status check operation and the validation use the same data structures.

The global write set is an ordered collection of the local write sets of the successfully committed transactions. The local write sets within the global write set are ordered by timestamp. The value of the timestamp is from the time when the transaction has committed and thereby has added its local write set to the global write set.

XPath queries XQ submitted by the client are collected in read sets. The server maintains one read set per mobile client. The read set is an ordered collection of XPath expressions XQ which have been sent from the client to

the server. A timestamp which indicates the last use of the XPath expression is associated to each XPath expression. The timestamp is updated every time a difference fragment for XQ is submitted to the client, caused by either a commit request, or a status check request operation, or an ordinary query.

Whenever there are too many fragments stored in the global write set, the server can displace some fragments starting with the fragments that have the oldest timestamp. As a consequence, the clients can not check the validity of their local data, if this local data has an older timestamp. Therefore, cached local data with a timestamp older than the oldest timestamp found in the global write set, is considered to be outdated.

4.2 Server-side status check operation

The server-side status check operation works as follows. Whenever a timestamp of a query received from the client is older than the oldest timestamp of a local write set stored in the global write set, then the fragment previously retrieved by this query is considered to be outdated and the complete new query result is transferred to the client. This is a special case in our approach, however note that the standard validation approach requires this amount of data transfer for each query.

Otherwise, the query is applied to each local write set fragment which is stored in the global write set and which has a timestamp that is newer than the timestamp of the query. Each XML fragment found in the global write set summarizes changes of the XML database caused by a concurrent transaction. The fragments describing changes of the XML database, which have occurred after the current transaction has read the database, are combined into a single difference fragment which is returned to the client as the result of the status check operation.

5. VALIDATION BASED ON XPATH QUERIES APPLIED TO GLOBAL WRITE SETS

Our synchronization protocol is adapted to the specific needs of mobile clients which must synchronize their server access and which should provide a reduction in the data transfer between client and server. Our protocol differs from the conventional parallel validation protocol contributed by Kung and Robinson [8], in various aspects. The most obvious differences are that our synchronization protocol extends validation with time stamps, works on XML fragments, is predicative, i.e. it applies XPath query expressions to

XML fragments, and is adapted to the exchange of XML difference fragments.

5.1 An optimistic protocol with read phase, validation phase and write phase

Lost connections and a long duration of transactions require a non-blocking transaction synchronization protocol. Our synchronization protocol is optimistic in the sense that transaction execution is performed on the client's cache within a read phase and that the success of the client's read phase depends on a commit decision or a status check decision made by the server.

Within the read phase, the client only reads data from the server (or the client's cache) and writes data into its local write set. Therefore, a transaction abort during the read phase will never damage any data within the server-side database.

There are the following alternatives for how the client's read phase is terminated. First, the client can abort the transaction with an explicit abort operation. This may happen as a result of a client application program error, or when the client decides that a connection has been lost for a period of time which is too long. Second, an abort stops a transaction when a status check request returns the result that the current client transaction has read outdated data, i.e. data which meanwhile has been modified by a successfully committed concurrent transaction. Third, a commit request of the client invokes a procedure commitRequest(...) on the server which terminates the transaction, and during which the read phase of the client is terminated, the validation is performed and eventually a write phase is performed.

5.2 The server-side procedure commitRequest(...)

The server-side procedure commitRequest is outlined in Algorithm 2 below. The parameters are the local write set of the transaction, the XPath expression XPwrite which describes the fragment accessed by write operations (insert, update, delete) and a list of pairs, each of which contains an XPath query expression and a timestamp. The timestamp associated with an XPath query XP denotes the server-time when the last difference fragment for XP has been computed (or the time when the result of XP has been computed if there was only one access to XP).

```
(1)  diffXMLfragment commitRequest( writeSet, XPwrite, list((XP1,t1),...,(XPn,tn)) )
(2)  { < tv = getValidationTimeStamp( ) > ;   // critical section for end of read phase
(3)    diffXML = validation( XPwrite, tv, list((XP1,t1),...,(XPn,tn)) ); //validation phase
(4)    if diffXML.isEmpty( )
          {  // write phase: apply insert, update & delete operations to server-side DB
(5)           XMLdatabase . applyChangesOf ( writeSet ) ;   // modify DB
(6)           globalWriteSet.add( writeSet, currentTime( ) ) ; // update global write set
(7)       }
(8)    < signalEndOfTransaction > ;      // critical section for transaction is completed
(9)    return diffXML ;                                  // inform client
(10) }
```

Algorithm 2: The server's implementation of commitRequest

The critical section within line (2) determines the end of the transaction's read phase and the beginning of the transaction's validation phase. As within the parallel validation contributed by Kung and Robinson [8], transactions are ordered according to their validation timestamp in the sense that newer transactions validate against older transactions, i.e. in case of a conflict the newer transaction is aborted and restarted.

The validation phase (line (3)) computes the difference XML fragment diffXML of outdated data as outlined below. If and only if this fragment is empty, the write phase, which consists of the following two parts, is performed. First, the client's modifications collected in the local write set are applied to the XML database (line (5)). Second, the local write set is added to the global write set, together with a timestamp (line (6)). Thereafter, the transaction is completed on the server-side (line (8)), and the resulting difference fragment is returned to the client (line (9)).

5.3 The predicative queries of the validation phase

Our validation protocol differs from the conventional parallel validation protocol contributed by Kung and Robinson [8], not only because we use XML fragments instead of database tuples for write sets and because we synchronize client transactions on a central server. Additionally, one key difference is that within our protocol, the server does not use sets of nodes or XML fragments as read sets, but instead uses the XPath expression submitted by the client in the read set. As the XPath expressions are usually considerably smaller than the read XML fragment, transferring these considerably smaller XPath expressions instead of the XML fragments allows a reduction in the data exchange from client to server. We consider this to be a competitive advantage in mobile clients that use small bandwidth connections to the server.

Our validation protocol applies XPath expressions of the validating transaction to XML fragments which have been collected in the write sets of older concurrent transactions. Similar to predicative validation [13], we use this as follows not only for read-write conflicts, but also for write-write conflicts.

With transactions To that are already completed when a validating transaction Tv enters its validation phase, Tv has to check only read-write conflicts, which is done as follows:

```
For each XPath expression XPE in the read set of Tv:
    For each modified XML fragment MXF in the write set of To,
        If timestamp(XPE) < timestamp(MXF)
        differenceXMLfragment . add( XPE applied to MXF ) ;
```

With an older transaction To2 that is validating concurrently to Tv (i.e. To2 enters its validation phase before Tv, but To2 is not completed at the time when Tv enters its validation phase), Tv has to check for write-write conflicts and for write-read conflicts. Since write expression set XPwrite (i.e. the second parameter of the procedure commitRequest(...) in Algorithm 2) contains all the XPath expressions which are used to write a fragment of the XML document, write-write conflicts can be checked together with write-read conflicts as follows:

```
For each XPath expression XPE in XPwrite or in the read set of Tv:
    For each modified XML fragment MXF in the local write set of To2,
        differenceXMLfragment . add( XPE applied to MXF ) ;
```

Note that all the computations of the validation can be performed with a usual XPath query evaluator, i.e. our approach does not need any additional tool. Additionally, this avoids the phantom problem, because conflicting insert and read operations are found by querying the inserted fragments. Note that the validation applies (read or written) XPath expressions to small modified XML fragments, i.e. not to the whole XML document.

Because the global write set consumes a limited memory, the oldest global write set entries are deleted when an overflow of this memory occurs. Therefore, a special treatment is provided for very old query results, i.e. query results that the client retrieved at a time t1 which was prior to the timestamp of the oldest entry that is still stored in the global write set. Whenever such an old query result has been used, it may be the case that global write set data has been deleted which is needed for validation. Therefore, validation regards these query results as outdated, i.e. validation fails and the results of these outdated queries or recomputed and transferred to the client.

5.4 Reducing the exchange of XML fragments for write operations of the client

When the client asks the server to commit a transaction, it transfers its local write set, i.e. a modified XML fragment containing only new values of inserted or updated nodes, to the server. As an additional parameter (the parameter XPwrite in Algorithm 2), the client sends an XPath expression that identifies the modified (inserted, updated or deleted) XML fragments to the server. Note however that it is not necessary that the client returns another modified XML fragment containing the old values of deleted or updated XML fragments to the server for two reasons. First, the XPath expressions sent to the server for the delete or insert operations are sufficient to identify those fragments of the server side XML document that have to be updated or deleted. Second, if the transaction validates successfully, the fragments to be updated or deleted have not been modified by a concurrent transaction. Therefore, the modified XML fragment containing old values can be computed simply by applying the XPath expression for delete or update to the server side XML document, just before the delete or update is applied to the XML document.

6. TREATMENT OF LOST CONNECTIONS AND CROSS TRANSACTION SYNCHRONIZATION

6.1 Lost connections do not stop a running commit request

Lost connections after the client's call of commitRequest(...) do not interrupt the validation process because all the data required for the validation phase and an eventual write phase are already stored on the server. As soon as the connection is re-established the server can inform the client about the result.

A lost connection before the call of commitRequest(...) can never damage data of any other transaction, because changes on the XML fragment are made on local copies on the client and are not yet transferred to the XML document.

Note furthermore, that a lost connection during a commitRequest operation never violates or blocks concurrent transactions, i.e. the only client which is prevented from continuing its work is the client that has lost its connection.

6.2 The client's repair options after re-establishing a lost connection

The client has the choice between four different options after a lost connection is re-established. First, the client can ignore its work, i.e. it can abort the client application. Second, the client an abort and restart the interrupted transaction. Third, the client may decide to continue the transaction as if nothing happened. Fourth, the client may use the *status check* operation in order to be informed whether or not it is useful to continue the transaction with the current content of the cache.

Which of the four alternatives is most appropriate after re-establishing a connection, depends on the work the client has done (i.e. if the client has not done much work, it may decide to restart the transaction) and on the duration for which the connection was lost (i.e. if the time was short, the client may decide to continue as if nothing happened).

Note that whatever a client decides to do after re-establishing a connection, no other client has to take care of whether or not the connection was lost and whether or not it was re-established during a running transaction. Furthermore, no other client can be damaged or delayed, which we consider to be an advantage of the optimistic approach that we use.

6.3 Cross transaction optimization

Whenever a client which has used server data during a previous successful transaction also requires this data in a following transaction, the client has similar options to those in the case of lost connections. The client can either use a status check in order to be sure that the data is still correct, before it starts further work, or the client can optimistically use the previous query result without any further server interaction. In the latter case, the client's previous query results are checked within the validation phase. Again, which of the decisions is appropriate, may depend on the time since the commit of the last transaction, on the work the client transactions will have to do, and on the probability that the data is changed by a concurrent transaction. For example, general data like a customer name is very unlikely to change. Therefore, it is reasonable to reuse the data even if it is stored in the client's cache with a very old time stamp, instead of reading it again within a new transaction.

6.4 Optimized restart of transactions

Furthermore, the same options as mentioned for a successor transaction also apply to the restart of a transaction. However, as restarts directly follow

an abort and because the difference XML fragment is returned to the client as the part of the commit decision, the client can integrate the difference XML fragment without an additional server access. Furthermore, the restarted transaction which operates on the modified difference XML fragment will most likely be successful (except for the rare case that the restarted transaction accesses a different fragment, or the case that other conflicting transactions perform their commit request operation in between).

Note however that in any case, the restart of a transaction requires significantly less data transfer from server to client.

7. SUMMARY AND CONCLUSIONS

We have presented a combination of client-side caching and optimistic transaction synchronization which treat lost connections and reduce the amount of data being exchanged between client and server through a variety of optimizations.

Our system can treat lost client connections during transaction execution without disturbing the work of other clients, i.e. it has the following properties which make it suitable for mobile clients. On the one hand, whenever a client loses its connection to the server, no other client is blocked or may retrieve outdated data. Even the client transaction itself can proceed as long as it does not require data from the server, however it cannot commit. On the other hand, when a lost connection is re-established, the client can choose between different options. First, the client can restart the whole application. Second, the client can abort and restart the actual transaction within the application. Third, the client can perform a status check in order to find out whether the data it had used is still valid. Fourth, the client can continue the transaction as if nothing happened, i.e. the validation at the end of the transaction checks whether or not the transaction has to be aborted and restarted.

Which option is the best, depends on different parameters, e.g. how long a connection was lost and how much work a transaction has done when the connection is re-established.

Furthermore, our approach to transaction synchronization and repairing lost connections integrates well with a reduction of data exchange between server and client. Wherever possible, the server computes and transfers only an XML difference fragment instead of submitting a complete XML fragment. Difference XML fragments are used within data status checks, within restarts of transactions in order to replace outdated data, and for ordinary queries. The computation and transferal of difference fragments instead of complete query results may be even more advantageous when the

XML data has to be generated on the server-side (e.g. by a transformer) from a different data source. In this case, difference queries can be used in order to reduce the amount of data which has to be generated.

Although we have presented and developed our approach specifically for the needs of mobile XML database clients that rely on a small bandwidth connection to a server, the approach seems to be equally appropriate for other mobile XML database and information systems using optimistic transactions as well.

REFERENCES

[1] Adya, A., Gruber, R., Liskov, B., Maheshwari, U.: Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks, ACM SIGMOD Int. Conf. on Management of Data, 1995.

[2] Stefan Böttcher, Adelhard Türling. Caching XML Data for Mobile Web Clients. International Conference on Internet Computing IC'04, Las Vegas, USA, Juni 2004.

[3] Stefan Böttcher, Rita Steinmetz. A DTD Graph Based XPath Query Subsumption Test. XML Database Symposium (XSym) at VLDB 2003, Berlin, September 2003.

[4] Chung, I.-Y., Hwang, C.-S.: Transactional Cache Management with Aperiodic Invalidation Scheme in Mobile Environments. ASIAN 1999: 50-61.

[5] Daniela Florescu, Alon Y. Levy, Dan Suciu: Query Containment for Conjunctive Queries with Regular Expressions. PODS 1998: 139-148.

[6] Gore, M.M., Ghosh, R.K.: Recovery of Mobile Transactions. DEXA Workshop 2000: 23-27.

[7] Ku, K.I., Yoo-Sung, K.: Moflex Transaction Model for Mobile Heterogeneous Multidatabase Systems. RIDE 2000: 39-46.

[8] Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control. ACM TODS, 6, 2, 1981.

[9] Amelie Marian, Jerome Simeon: Projecting XML Documents, VLDB 2003.

[10] Gerome Miklau, Dan Suciu: Containment and Equivalence for an XPath Fragment. PODS 2002: 65-76.

[11] Frank Neven, Thomas Schwentick: XPath Containment in the Presence of Disjunction, DTDs, and Variables. ICDT 2003: 315-329.

[12] Öszu, M.T., Valduriez, P.: Distributed Database Systems, 2nd Ed., Prentice Hall, 1999.

[13] Reimer, M.: Solving the Phantom Problem by Predicative Optimistic Concurrency Control, 9th VLDB, Florenz, 1983.

[14] XML Path Language (XPath) Version 1.0 . W3C Recommendation November 1999. http://www.w3.org/TR/xpath.