USENIX Association

Proceedings of MobiSys 2003:
The First International Conference on
Mobile Systems, Applications, and Services

San Francisco, CA, USA
May 5-8, 2003

Reservations for Conflict Avoidance in a Mobile Database System *

Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, Henrique Domingos
Departamento de Informática
FCT, Universidade Nova de Lisboa, Portugal

Abstract

Mobile computing characteristics demand data management systems to support independent operation. However, the execution of updates in a mobile client usually need to be considered tentative because uncoordinated updates that conflict need to be reconciled. In this paper we present a mechanism to independently guarantee that updates can be executed in the server without conflicts. To this end, clients obtain leased reservations upon the database state. Updates are specified as common small PL/SQL programs, dubbed mobile transactions, that execute both in the mobile client and in the server. Using the available reservations, the client transparently verifies that a transaction can be executed in the same way both in the mobile client and in the server, thus leading to the same final result. Mobile transactions may specify conflict detection and resolution rules to be used when transactions cannot be locally guaranteed.

1 Introduction

Despite advances in hardware and communication technology, the connectivity in mobile devices is intermittent because of physical, economical and energy factors. These characteristics call for support of independent operation, i.e., users must be allowed to read and modify a shared database with no remote synchronization. Even when connectivity is available, independent operation may be used to overcome latency and bandwidth problems, interference among multiple long interactive transactions and to reduce load during service peaks.

Optimistic replication is used to support independent operation [19, 14, 20, 11]. In such approaches, clients maintain local data copies and modify them independently. The uncoordinated updates are integrated in the common database state solving any conflict caused by concurrent modifications. As this reconciliation process may lead to a different final result (or even to the abortion of the transaction), the client can not independently determine the final result of a transaction.

This paper presents the Mobisnap mobile database system, focusing on support for independent operation. The Mobisnap middleware extends a traditional client/server SQL database system. The single server maintains the *official* database state. Mobile clients cache database snapshots. Applications running on mobile devices update the shared database submitting small PL/SQL [13] programs, dubbed *mobile transactions*. During independent operation, mobile transactions are tentatively executed in the mobile clients. Later, the clients propagate these mobile transactions to the server, where the transaction programs are re-executed against the *official* database state.

Unlike transaction re-execution and validation based on *read and write sets*, the execution of the mobile transaction program in the server allows the definition of conflict detection and resolution rules that exploit the semantics of the operation (a similar approach is used in Bayou [20]).

Mobisnap combines this conflict resolution strategy with a conflict avoidance mechanism based on *reservations*. A reservation provides some kind of promise upon the database state, depending on the type of reservation. Mobile clients may obtain leased [5] reservations before disconnecting. When a mobile transaction is executed in the client, the system transparently verifies if the available reservations are sufficient to guarantee its final result independently. If the client holds enough reservations, the local result can be considered definite because reservations guarantee that no conflict will arise when the transaction is re-executed in the server.

Our reservation mechanism presents the following contributions. First, it includes several types of reservations (unlike other proposals that define a single type, such as escrow techniques [9]). As it is shown in section 4.5, it is usually necessary to combine the use of different types of reservations to guarantee the result of any transaction. Second, the system transparently verifies if the available reservations can guarantee the result of a mobile transaction written in unmodified PL/SQL (unlike previous systems [12], no special function is used to access the reserved data items). Third, it implements the reservation model integrated with mobile transactions in a SQL-based system.

The Mobisnap implementation described in this paper represents an interoperable and evolutionary middleware approach towards mobility, instead of a "revolutionary" one. Application that run in the Mobisnap system can use the new mechanisms – mobile transactions and reservations. However, legacy clients are still allowed to access the database server directly, without modifications.

This paper is organized as follows. Section 2 discusses the design principles in the context of typical applications. Section 3 presents the overall Mobisnap system. Section 4 details reservations. Section 5 describes the current status of our work. Section 6 presents an evaluation of reservations. Section 7 discusses related work and section 8 concludes the paper with some final remarks.

2 Design principles

The main concepts of the Mobisnap system will be highlighted with the help of three typical scenarios — we will come back to these examples throughout the paper. In the first scenario, the user is a mobile salesman selling commodities (e.g., CD's, shoes, etc.). The number of commodities can be large and the available stock for each one is limited — all items of a given commodity are assumed identical. The salesman receives orders from her customers that must be satisfied with the current stock.

The second scenario is a variant of the first one, but the commodity's items are not identical (e.g., tickets for the theater, train tickets, etc.).

In the third scenario, the database contains a datebook with appointments. Several persons can change the same datebook (e.g. a person and his secretary). The operations typically insert or remove an appointment.

In all the above scenarios, the mobile database system must support independent operation to allow users to continue their work while disconnected. During independent operation the system can provide local access to shared data using partial database snapshots. For example, a salesman only needs to cache information about the products he sells and the customers he intends to visit.

2.1 Mobile transactions, a tool for handling concurrent updates

A transaction executed independently in a mobile device observes the cached database state. Later, when the transaction is propagated to the server, other transactions might have changed the database. In this case, the transaction execution in the client, as defined by its *read and write sets*, might no longer be valid. To solve these situations it is necessary to rely on semantic information [20, 11] to define appropriate conflict detection and resolution rules that must be enforced when transactions are integrated in the *official* database state.

In Mobisnap, operations that modify the database are expressed as *mobile transactions* (or simply transactions where no confusion may arise). A mobile transaction is a small program written in a subset of the PL/SQL [13] language. The following changes to PL/SQL have been introduced: (1) *commit* and *rollback* statements end the execution of a mobile transaction, i.e., they act as a return; (2) when the result of a *select into* statement includes more than one (sub-)row, one row is returned (instead of raising an exception); (3) the *newid* function was added to return the same new unique identifier when the transaction is executed in the client and in the server.

A mobile transaction is executed tentatively in the client, and later propagated to the server. Its final result is only obtained when the transaction is executed in the server. A mobile transaction is always executed running its program with a specially designed PL/SQL interpreter. Each mobile transaction accesses the database in the context of a distinct database transaction.

Programmers may reason about a mobile transaction as a mobile program that will be executed in the server. This approach allows programmers to express suitable conflict detection and resolution rules for each situation. The following examples, from the previous application scenarios, show two possible approaches. The mobile transaction of figure 1 inserts a new order received by a mobile salesman. In this case, the conditions for accepting the order are precisely expressed — stock availability and price acceptability are checked (line 7) before inserting the order (lines 8-11). The transaction is not aborted if the values are different in the server, but only if the specified conditions are violated. The mobile transaction of figure 2 inserts a new appointment in a shared calendar giving two alternative periods of time (lines 3-4 and 8-9 check the calendar availability for each alternative). In this case, conflicts can be solved if, at least, one of the expressed alternatives is possible.

2.2 Reservations, a tool for guaranteeing results independently

As it has been explained before, the independent execution of a transaction must be considered tentative. However, independently guaranteeing the result of a transaction might be important. For example, a mobile salesman could immediately guarantee that the orders received can be satisfied. To this end, locks are the traditional solution, but they must be adapted to be used in mobile environments. As a mobile client must keep the locks while it is disconnected, locks should affect as few data items as possible. Sometimes locks are still unnecessarily restrictive. For example, escrow techniques [9] can guarantee concurrent updates to variables that represent partitionable resources – the available resources are divided among the mobile clients.

Any of these solutions alone is insufficient to guarantee

```
1    -------- ORDER PRODUCT: name = "BLUE THING"; quantity = 10; highest price = 50.00 ------
2    DECLARE
3      prd_price FLOAT;
4      prd_stock INTEGER;
5    BEGIN
6      SELECT price, stock INTO prd_price, prd_stock FROM products WHERE name = 'BLUE THING';
7      IF prd_price <= 50.00 AND prd_stock >= 10 THEN        -- checks price acceptability and stock availability
8        UPDATE products SET stock = prd_stock - 10 WHERE name = 'BLUE THING';
9        INSERT INTO orders VALUES (newid,'Clt foo','BLUE THING',10,prd_price);   -- newid returns the same unique identifier
10       NOTIFY( 'SMTP', 'sal-07@thingco.pt', 'Order accepted ...');            -- in the client and in the server
11       COMMIT prd_price;                                  -- commits and returns the price used in transaction
12     ENDIF;
13     ROLLBACK;                                            -- rollbacks and returns when customer's preferences cannot be satisfied
14   ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Impossible order...');    -- sends notification to user on rollback
15   END;
```

Figure 1: Mobile transaction specifying a new order submitted by a mobile salesman.

```
1    -----------SCHEDULE MEETING: '17-FEB-2002' at 10:00 OR '18-FEB-2002' at 9:00 --------------------------------
2    BEGIN
3      SELECT count(*) INTO cnt FROM datebook WHERE day='17-FEB-2002' AND hour=10;       -- first alternative
4      IF (cnt = 0) THEN                                  -- checks calendar availability for 17-FEB-2002 at 10:00
5        -- code omitted: update datebook, send notification if appropriate, ...
6        COMMIT ('17-FEB-2002',10);                       -- commits and returns info on the committed alternative
7      ENDIF;
8      SELECT count(*) INTO cnt FROM datebook WHERE day='18-FEB-2002' AND hour=9;        -- second alternative
9      IF (cnt = 0) THEN                                  -- checks calendar availability for 18-FEB-2002 at 9:00
10       -- code omitted: update datebook, send notification if appropriate, ...
11       COMMIT ('18-FEB-2002',9);                        -- commits and returns info on the committed alternative
12     ENDIF;
13     ROLLBACK;                                          -- rollbacks and returns when no alternative is possible
14   ON ROLLBACK NOTIFY( 'SMS', '351927435456', 'Impossible to schedule... ');
15   END;
```

Figure 2: Mobile transaction adding a new appointment to a shared calendar (declaration of variables is omitted).

the outcome of the mobile transactions presented in figures 1 and 2 (without being too restrictive). Consider the example of figure 1. To guarantee that the order can be accepted, it is necessary to guarantee that the conditions expressed in line 7 are true when the transaction is executed in the server. Escrow techniques can guarantee stock availability ($prd\_stock \geq 10$), but they do not help to guarantee price acceptability ($prd\_price \leq 50.00$). In this case, a lock can be used but it is too restrictive (a reservation that allows to use a reserved value for a data item, despite of its current value, can be used in our system, as described in section 4). Application knowledge, external to the system, can also be used to guarantee the validity of some conditions by voluntarily limiting the degree of concurrency (e.g. prices are only updated during the night). This approach is not safe and it can be easily broken in a large scale scenario. Therefore, the system needs to combine different techniques in a single framework to safely guarantee the outcome of a transaction independently (see other examples in section 4.5).

In Mobisnap, the outcome of mobile transactions can be guaranteed independently using an SQL-based *reservation* mechanism (detailed in section 4). A reservation is similar to a semantic lock and it provides some promise upon the database state. Mobisnap defines several types of reservations, each one providing a different type of promise. The different types of reservations were introduced to support the typical scenarios described earlier. Reservations are valid during a limited period of time

(thus preventing a client that becomes permanently disconnected from holding a reservation forever).

Mobile clients obtain reservations before disconnecting. When a transaction is executed in the client, the system transparently checks that the client holds enough reservations to prevent other clients from making updates that might later be found to conflict with the local transaction. If enough reservations exist, the client can independently guarantee the final result of the transaction because reservation guarantee that the transaction program is executed in the same way both in the client and in the server (given that it is propagated to the server before the used reservations expire).

Our reservation model transparently uses all available reservations to guarantee mobile transactions (i.e., the transaction do not need to specify which reservation should be used). Thus, no special function is needed to manipulate the reserved data items and programmers can specify operations as normal PL/SQL programs.

Some ideas used in our reservation model had already been proposed [5, 9, 7]. However, the integration of multiple types of reservations (including the introduction of new ones) and their adaptation to mobile environments, the transparent detection of their usage and its implementation in an SQL-based system are new contributions that are important for the usability of these concepts. As far as we know, its integration with mobile transactions makes our system unique.
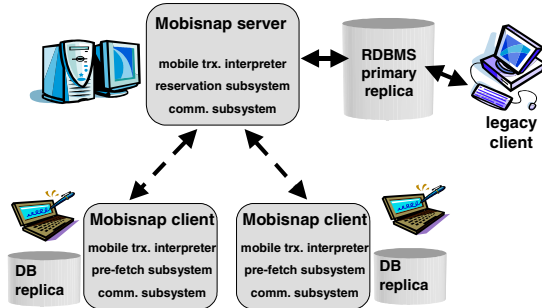
Figure 3: The Mobisnap system architecture.

3 Mobisnap system

The Mobisnap system manages information structured according to the relational data model. Its architecture is based on an *extended client/server* middleware architecture, as depicted in figure 3. Mobisnap server and clients rely on unmodified relational database systems to store data. The server holds the primary copy of all data items in the central database. The server is expected to be mostly available and well connected. Legacy clients can access the central database directly (i.e., without accessing the Mobisnap server).

Mobisnap clients (or simply clients) can run on mobile or stationary computers and be disconnected for long periods of time. Mobisnap clients do not access the central database directly – they always communicate with the Mobisnap server. Users access the database using applications that run on client machines. When connected, the read and write operations executed by an application may be immediately executed in the server.

We now describe independent operation (and the mechanisms to support it). A mobile client locally replicates a subset of the database. This partial replica contains a subset of the database tables; for each table a subset of the columns; finally, only a subset of the records is cached for each table. Currently, the user/application defines the cache contents using a variant of the SQL *select* statement. Other data accessed by the application is also cached using a least recently used algorithm. The prefetch subsystem in the client periodically updates cache contents. This simple strategy could be improved using more complex caching techniques (e.g. [3, 10]), but this problem is outside of the scope of this paper.

The client maintains two copies of the database state: a tentative and a committed one. The tentative version contains the state after executing all disconnected transactions. The committed version contains the state after executing transactions that are guaranteed by reservations. This version is only maintained if enough resources exist. Both versions are refreshed when a mobile clients connects to the server by obtaining, at least, a new copy of the modified records. Applications may read either ver-

sion using SQL queries — the application selects which version should be used for each query.

Applications modify the database by submitting mobile transactions. The client executes each mobile transaction locally and returns a result. If the client holds enough reservations to guarantee the outcome of the transaction, the result can be considered definite. In this case, both versions of the cache are updated. Otherwise, the result is tentative – only the tentative version is updated.

After being locally executed, a mobile transaction is logged in the mobile device. When the mobile client reconnects, it propagates logged transactions to the server. On weakly-connected environments, propagation may be incremental.

When the server receives a mobile transaction, it executes the transaction program to obtain its final result. If the transaction has been guaranteed in the client and all used reservations are still valid, the reservation model guarantees that the outcome of the transaction is the same in the client and in the server. In the next section we detail the reservation model and its interaction with transaction processing.

When a client obtains a reservation, the server must guarantee that no transaction from any other client violates the promise provided by the reservation. For example, if a client reserves one seat in a train (to guarantee transactions independently), the server guarantees that, at least, one seat remains available – any transaction that tries to book the last available seat is aborted. If the reservation is not used, some transactions may have been aborted unnecessarily – in the end, one seat is available.

To handle these situations, Mobisnap includes a reevaluation mechanism. Transactions that abort in the server can be re-executed after relevant reservations are cancelled or expire. In this sense, a reservation can be seen as an option to modify the database first. When an application submits a mobile transaction, it may specify that the reevaluation mechanism should be used and set the deadline to obtain the final result, i.e., the last time for transaction re-execution in the server.

When the final result of a mobile transaction is obtained, the user may no longer be connected to the system. In Mobisnap, mobile transactions may use the *notify* function to immediately notify users of its final result (e.g. using SMS or pager messages). The *notify* invocations are handled in a special way: they are only executed during the final execution of the transaction in the server. The messages produced by the *notify* function are propagated asynchronously by the Mobisnap server (that handles temporary errors). A similar approach can be used to defer other actions on the outside world until the definite execution of a transaction, thus avoiding the need to undo the effects of tentative actions.

4 Reservations

The goal of reservations is to independently guarantee the final result of mobile transactions by guaranteeing that no conflict will arise when the transaction program is executed in the server. The Mobisnap reservation model is designed to achieve this goal while applications continue to use traditional SQL statements.

4.1 Types of reservations

A reservation provides some kind of guarantee for the future execution of a transaction in the server. In this subsection we present the types of reservations implemented in Mobisnap (see section 4.5 for examples).

Value-change and slot reservations

A *value-change reservation* provides the exclusive right to modify the state of an existing data item (i.e., a subset of columns in some row). For example, a user may reserve the right to change the description of a specific seat in a train. This is like a traditional fine-grain write lock [7].

A *slot reservation* provides the exclusive right to insert/remove/modify rows with some given values. For example, a user may reserve the right to schedule a meeting in a given room in some defined period. This is like a predicate lock [7], and includes rows that exist and that do not exist.

The granularity and lease time of these reservations must be selected carefully because they prevent other transactions from modifying the selected data. In this sense, these reservations can be seen as a mechanism to move the primary copy of a data item to a mobile device.

Value-use reservation

A *value-use reservation* provides the right to use a given value for some data item (despite its current value). The need for this kind of guarantee is common in real life – e.g. a mobile salesman can guarantee a price for an order despite concurrent changes. This reservation implements this idea, allowing transactions to guarantee the value of a data item without unnecessarily restricting concurrent changes. Transactions that use this reservation *observe* the reserved value of the data item (*select* statements return the reserved value instead of the current value).

Escrow reservation

An *escrow reservation* provides the exclusive right to use a share of a partitionable resource represented by a numerical data item. For example, the stock of some product may be split among several salesmen. This reservation is based on the escrow model [12, 9].

The escrow model is applied to numerical data items that represent partitionable resources, where all items are identical — e.g. the number of CDs in the stock, x (in section 4.5, we show how reservations can address problems with non-identical resources – e.g. seats in a train). The following properties usually hold for this type of data items. First, they are updated by adding/subtracting a constant value — e.g. when k CDs are sold, we do $x \leftarrow x - k$. Second, some constraints must be maintained — e.g. the stock must be larger than *min*, i.e., $x \geq min$.

For such data items (called escrowable data items), it is possible to partition the resources among several independent replicas — e.g. let x be partitioned in n parts $x_i, 1 \leq i \leq n$, such that $x = \sum x_i$. Each replica has an associated local constraint that guarantees the validity of the global constraint, $x_i \geq min_i : min = \sum min_i$. Each replica may independently guarantee the result of transactions that comply with the local constraints, i.e., it is possible to guarantee the result of transactions that subtract up to $x_i - min_i$ (aggregated value over all transactions in replica i) to x_i.

Example: Let the current stock of CDs be ten ($x = 10$) and the minimum stock be two ($x \geq 2$). Using the escrow model, the current stock can be partitioned between two replicas – e.g. the first replica gets six CDs ($x_1 = 6$) and the second replica gets four CDs ($x_2 = 4$). The global constraint must also be partitioned between the replicas – e.g. the stock must be greater or equal to one in both replicas ($x_1 \geq 1 \wedge x_2 \geq 1$). The first (resp. second) replica can guarantee transactions that subtract up to five (resp. three) CDs from the stock ($x_1 - min_1 = 6 - 1 = 5$ and $x_2 - min_2 = 4 - 1 = 3$).■

We now discuss the Mobisnap implementation of the escrow model. To recognize the aspects involved, consider the example of figure 1. In this mobile transaction, escrow techniques can be used to guarantee the stock availability. The following steps are taken to subtract a constant to the stock (these steps represent a pattern to access escrowable data items): (1) the current value is read and the validity of the operation is checked (lines 6–7); (2) the value of the data item is updated to reflect the executed operation (line 8).

The first aspect to address is to guarantee the validity of the operation (step 1). To this end, it is usual to use an *if* statement that verifies if the update violates the local constraints or not. To use the same condition without any special function, the same local constraints must be defined in the server and in the client. The escrow model, as defined before, cannot be implemented immediately as it uses different constraints in different replicas.

In Mobisnap,we have introduced the following change to the model presented before: $x : x \geq min$ is partitioned in n parts $x_i : x_i \geq min \wedge x - min = \sum(x_i - min)$. As before, each replica may independently guarantee the result of transactions that comply with the the local constraints. However, the same constraints should be enforced in all

replicas, thus allowing mobile transactions to verify the validity of an update using the well-known global constraints (or, even, using more restrictive constraints).

Example: To obtain the same guarantees as in the previous example, the following values are used: $x_1 = 7, x_1 \geq 2$ and $x_2 = 5, x_2 \geq 2$. The first (resp. second) replica can guarantee transactions that subtract up to five (resp. three) CDs from the stock ($x_1 - min = 7 - 2 = 5$ and $x_2 - min = 5 - 2 = 3$).

Let x_1 be the server replica and x_2 be the mobile client replica. $x_2 = 5, x_2 \geq 2$ means that the mobile client has reserved the right to subtract 3 to x (i.e., to guarantee orders for 3 CDs). We say that the mobile client has obtained an escrow reservation for three instances of x. In the server, the value of x is updated ($x_1 = 10 - 3 = 7$) to reflect the guarantees obtained by the client. Thus, no transaction from any other client can use the reserved resources. Alternatively, the value of x_2 can be seen as the minimum value of x when the transactions from the mobile client are executed in the server (i.e., $x \geq x_2$). Therefore, the value of x_2 can only be used to guarantee conditions that use similar relational operators – e.g. the condition $x \leq 10$ can not be guaranteed by the escrow reservation $x_2 = 5, x_2 \geq 2$. The value of x_1 can be seen as the value of resources not reserved by any mobile client (and thus available, for example, to legacy clients).■

The second aspect to address is the *real* update of the escrowable data item (step 2). Usually, mobile transactions update these data items using the *update* statement. In Mobisnap, the system automatically infers the amount of reserved resources used from the *update* statements. As expected, the used resources are consumed from the escrow reservation, i.e., the following transactions can only be guaranteed by the remaining reserved resources.

As described, our approach is completely transparent for programmers. Programmers write mobile transactions with no reference to reservations. If the client holds some reservations, the system automatically uses them to guarantee the result of transactions. The system also keeps track of used reservation automatically.

Our current prototype has a limitation: it only allows a single constraint over each escrowable data item (either $x \geq min$ or $x \leq max$) — we anticipate this to be the typical scenario.

Shared value-change and shared slot reservations

A *shared value-change reservation* provides the guarantee that it is possible to modify the state of an existing data item (i.e., a subset of columns in some row). For example, this reservation can be used to guarantee increment operations to a shared counter.

A *shared slot reservation* provides the guarantee that it is possible to insert/remove/modify rows with some given

	vc	s	vu	e	shvc	shs
value-change (vc)	no	no	yes	no	no	no
slot (s)	no	no	yes	no	no	no
value-use (vu)	yes	yes	yes	yes	yes	yes
escrow (e)	no	no	yes	yes*	no	no
shared value-change (shvc)	no	no	yes	no	yes	yes
shared slot (shs)	no	no	yes	no	yes	yes

* Yes, while aggregate reservations do not violate global constraint

Table 1: Reservation compatibility table.

values. For example, this reservation can be used to guarantee operations in append-only tables.

These reservations do not provide any promise about the future state of the database. Instead, they prevent other clients from setting exclusive reservations that forbid the execution of the operations. As the examples of section 4.5 show, these reservations are important to fully guarantee the result of a transaction.

4.2 Granting and enforcing reservations

We now describe how reservations are granted and what is done to guarantee that no transaction violates those reservations (including transactions executed directly in the database by legacy clients). In the next subsections, we present the mobile transaction processing in the client and in the server when reservations are used.

A mobile client requests a set of reservations from the server before disconnecting. This set must be defined based on the operations the user is expected to execute until the next interaction with the server. The deduction of good values for this problem can be seen as an extension of the cache hoarding problem [10] (where clients must pre-fetch the data that will be accessed by users) and it is out of the scope of this paper. To solve this problem, forecasting techniques [4] can be used.

In the experiments presented in section 6 we use simple strategies that lead to good results in a mobile sales application. The GUI of our sales application allows to modify the reservations to obtain using a simple form. Internally, reservation requests are submitted using variants of the SQL select statement (e.g. *get value-use reservation price from products where id='5'* requests a value-use reservation for the *price* of product with *id='5'*).

When a reservation is requested, the server checks if it is possible to grant the reservation. First, it verifies if there is no other granted reservation that conflicts with the request. Table 1 presents the compatibility of two reservations that overlap, where *yes* means that it is possible to grant a reservation of a certain type even if other reservations of the other type already exist for the same data item. If data items do not overlap, reservations do not conflict. Second, it verifies if the user of the mobile client can obtain the requested reservations. This verification is based on authorization rules, set by the database administrator, that specify which reservations each user can obtain and for how long.

When a reservation is granted, the Mobisnap system has to guarantee that its underlying promise is not broken by other transactions. To enforce the promises, while allowing legacy clients to directly access the central database, the following actions are executed on the database (note that these actions are reversed to allow the execution of mobile transactions that use the given reservation):

- For each *value-change reservation*, a trigger is set to prevent transactions from modifying the reserved data item. A trigger prevents an update by throwing an error. We say that the update was blocked.
- For each *slot reservation*, a trigger is set to prevent transactions from inserting/deleting/modifying rows with the given values.
- For *escrow reservations*, the value of the escrowable data item is updated as explained earlier.
- For *value-use, shared slot and shared value-change reservations*, no action in the database is needed.

For value-change and slot reservations there is an additional option that can be used: the reservation request may specify a temporary update to reflect the impossibility to change the reserved data item while the reservation is valid. For example, assume that a row represents a seat in a train. When a mobile client obtains a value-change reservation over that row, the reservation may temporarily set the column that indicates if the seat is occupied to true in the server. This change indicates that no unguaranteed transaction should use that seat. If they do, the trigger aborts them, as in the normal case.

The Mobisnap system keeps track of all granted reservations to check conflicting reservation requests and validity of reservation usage. When reservations expire or are cancelled explicitly by the mobile client, the actions executed to enforce reservations are undone.

As it has been said, a reservation is leased [5], i.e., it is only valid for a limited period of time. This property guarantees that restrictions imposed to other transactions to enforce the promises associated with a reservation do not last forever, even if the mobile client that holds the reservation is destroyed or becomes permanently disconnected. On the other hand, the guarantee provided in a mobile client respecting some mobile transaction is only valid if the transaction is propagated to the server before the used reservations expire.

4.3 Transaction processing in the client

The mobile client executes a mobile transaction in, at most, two steps. In the first step, the system executes the transaction program verifying if its outcome can be guaranteed by the available reservations (see details next). If it can, the committed and the tentative database versions are updated. If not, the system rollbacks any change to the database and proceeds to the second step.

In the second step, the system executes the transaction program against the tentative database version. The result of this execution is considered tentative. If the program runs until a *commit* statement, the result is *tentative commit*. If the program runs until a *rollback* statement, the result is *tentative abort*.

If during transaction processing (in both steps) some non-cached data item is needed to proceed (e.g., an *if* statement uses the value of a non-cached column), the execution is aborted, and the result is *unknown*.

Verifying if a transaction can be guaranteed

Now, we outline how the client interpreter verifies if a mobile transaction can be guaranteed. The basic idea consists in running the transaction program and verify every statement in the execution flow.

For each variable, the interpreter maintains not only its current value but also the information if the value is *guaranteed* and the reservations that guarantee it (if any). The value of a *variable is guaranteed* if it was set in a guaranteed SQL read statement, or in an assignment statement that does not include any unguaranteed variable.

An SQL statement can be guaranteed if: (1) the variables used in the SQL statement (as inputs), if any, are guaranteed; (2) there is a reservation that is compatible with the condition expressed and includes the read/written columns. This second prerequisite is verified comparing the semantic description of the reservation with the conditions expressed in the SQL statement (a similar idea is used in semantic caching [3]). The following additional restrictions apply. A *value-use* reservation cannot guarantee an *SQL write statement* (*update*, *insert* or *delete*). A shared reservation can not guarantee an *SQL read statement* (*select*).

When an *SQL write statement* is executed, both database versions are updated. A guaranteed *SQL read statement* returns the reserved value. An unguaranteed *SQL read statement* returns the value of the tentative database version.

An *if statement is guaranteed* if all variables involved in the *if* condition are guaranteed (the restrictions explained in section 4.1 apply to variables guaranteed by escrow reservations). If the condition cannot be guaranteed, its value is assumed false, by default. Thus, it is possible to guarantee an alternative update (in a sequence of alternative updates guarded by *if* statements) when the preferred one cannot be guaranteed. For example, in the transaction of figure 2, if the client only holds reservations for the day "18-FEB", it is impossible to guarantee the first alternative (day "17-FEB") but it is possible to guarantee the second one (day "18-FEB"). When the transaction is re-executed in the server, the preferred alternative may be possible. By default, the server executes the same execu-

tion path, despite the current value of the database. The application may override the default options and request either to abort client execution if a condition cannot be guaranteed or to execute the first possible alternative in the server, instead of the guaranteed one.

If the program runs until a commit statement, its result is *reservation commit*. In this case, the transaction result is said locally guaranteed. However, depending on the statements that have been guaranteed, the following levels of guarantees are provided.

- *Full*, if all statements in the execution path are guaranteed by reservations. Note that even in this case, it is incorrect to apply, in the server, the *write set* obtained during the client execution because escrowable data items have to be updated using add/remove operations.
- *Read*, if all statements but write statements are guaranteed by reservations. When the transaction is executed in the server, non-guaranteed writes may be blocked by slot or value-change reservations.
- *Pre-condition*, if all executed conditions (*if* statements) are guaranteed. In this case, the system only guarantees that the execution of the transaction program in the server will follow the same execution path.
- *Alternative pre-condition*, if, at least, one condition (*if* statements) could not be guaranteed. In this case, the application has the option to force the same execution path or not (as explained earlier).

As exemplified in section 4.5, an application should use additional domain-knowledge to interpret the meaning of these levels of guarantees (e.g. the *read* and *full* levels are identical if it is known that no reservation will be granted that block the transaction's writes) and to present the result of transactions to users.

If the mobile transaction runs until an *rollback* statement, the system rollbacks the execution and proceeds transaction processing in the second step described in the beginning of this subsection.

When the result of a transaction is *reservation commit*, the mobile client automatically associates the information about program execution (reservations used and the execution path) with the transaction. The client propagates this information to the server. The server uses this information when it executes the transaction, as explained in the next subsection.

4.4 Transaction processing in the server

The execution of a mobile transaction in the server consists in the execution of the transaction program against the central database. If the transaction was not guaranteed in the client, user's intents must be enforced by the conflict detection and resolution rules specified in the code of the transaction. Blocked write statements can be handled by trapping the thrown error in the code of the transaction. Otherwise, the transaction is aborted and the application may check the cause. Aborted transactions may be re-executed using the reevaluation mechanism.

If the transaction has been guaranteed in the client and it has associated reservations, the interpreter that runs the mobile transaction must execute the following additional actions. Before starting to run the program, the system undoes the actions that enforce the used reservations (see section 4.2). For each value-use reservation, the current value of the data item is replaced by the reserved value – when the execution of the transaction ends, the current value is restored in the database.

During the execution of the transaction, for any read statement that was guaranteed in the client by a value-change or slot reservation, the interpreter returns the same value read in the client. This property guarantees that the same rows are read when the solution for a read statement is a set of rows (i.e. it establishes an order on the result set).

The execution of a mobile transaction partially consumes escrow reservations (e.g. if the client had the right to subtract 3 to x, and this transaction subtracts 1 to x, the client remains with the right to subtract 2 to x). All other reservations remain valid. After the execution of the transaction, the system redoes the actions that enforce the used reservations that remain valid.

4.5 Examples

The mobile transaction presented in figure 1 represents a typical transaction executed by a mobile salesman: the insertion of a new order. In figure 4 we present the reservations obtained by the mobile salesman. The following information is maintained for each reservation: a unique identifier; the *type* of reservation; the *columns* that are reserved in the rows identified by the *table* and *condition*; the *value* of the reserved data item; and additional reservation-specific information.

When the transaction is executed in the client, it is obvious that both reservations are necessary to guarantee that the condition expressed in the *if* statement (line 7) is true. The escrow reservation guarantees that $prd\_stock \geq 10$ will be true when the transaction is executed in the server ($prd\_stock_{clt} = 15$, thus $prd\_stock \geq 10$ evaluates to $15 \geq 10 \equiv true$). In fact, the reservation is more general and it gives the right to subtract 15 to the value of prd_stock with the global constraint $prd\_stock \geq 0$ (or the promise that when the transaction is executed in the server, $prd\_stock \geq 15$). The value-use reservation guarantees that the price is acceptable. The update of the stock (line 8) is guaranteed by the escrow reservation.

id	type	table	column	condition	value	info
45-1	*escrow*	products	stock	name='BLUE THING'	15	≥ 0
45-2	*value-use*	products	price	name='BLUE THING'	44.99	-

Figure 4: Reservations obtained by a mobile salesman.

This statement is used to infer the escrowable resources consumed by the transaction — 10 in this case (the number of resources still available after the execution of this transaction is 5 — this is the new value of the reserved data item (*products*, *stock*)). The *insert* statement (line 9) is not guaranteed by any reservation. Thus, the transaction will have the *read* level of reservation commit. If the application knows that no slot reservation will block the *insert* in the "orders" table, it can be sure that the transaction will succeed in the server.

It is also possible to get additional reservations that guarantee the success of the *insert* statement. First, a shared slot reservation could be obtained for the orders table (uniqueness of identifier in the table is assured by the properties of the *newid* function). Second, an additional column could be added to the orders table specifying the salesman responsible for the order. In this case, the salesman could obtain a slot reservation for his orders.

The mobile transaction presented in figure 5 represents a typical scenario for using non-identical resources. In this example each seat is unique. Therefore, after verifying that there is, at least, one seat left (line 5), it is necessary to obtain the identifier of the seat (lines 7-8) and update the seat information with the name of the new passenger (lines 9-10). Assume that two seat are reserved: the correspondent reservations are presented in figure 6. Up to the need of obtaining the identifier of the seat, the transaction processing can be guaranteed as in the previous example. The *select* statement (lines 7-8) that obtains the seat returns one of the reserved seats (e.g. "4A"). The *update* statement (lines 9-10) updates the seat information with the name of the passenger and the price of the ticket. These statements are guaranteed by the value-change reservation obtained over the seat (remember that, in the client, read statements return reserved data items, if any exists). When the transaction is executed in the server, Mobisnap guarantees that the *select* statement returns the same record (i.e. seat "4A"). These reservations guarantee the *full* level of reservation commit (all statements are guaranteed).

If the value-change reservation was not available, it would be impossible to guarantee the read and write statements of lines 7-10. In this case the transaction would have the *pre-condition* level of reservation commit. For this transaction, this result means it is possible to guarantee the availability of one ticket but, in the client, it is impossible to know which seat will be assigned to the passenger (the *select* statement of line 7-8 would return a tentative result based on the tentative cache version).

The mobile transaction presented in figure 2 represents a typical transaction executed in a shared calendar. The client may obtain a slot reservation to schedule appointments during the morning of day '17-FEB-2002', as shown in figure 7. This reservation guarantees the result of the condition of line 4. The variable *cnt* is guaranteed by the slot reservation because the records selected in line 3 are a subset of the records associated with the slot reservation. The code omitted in line 5 would typically insert a record in the datebook for the given time – this operation can also be guaranteed by the slot reservation. Thus, the result of the transaction would be the *full* level of reservation commit.

These examples show that the combination of multiple reservations is fundamental to guarantee the result of a mobile transaction independently. Note that not even the example that uses anonymous resources (figure 1) can be guaranteed by escrow reservations alone.

5   Status and future work

We have implemented a prototype of the Mobisnap system as a middleware layer in Java. In the server, an Oracle 8 database stores the primary copy of the data, but any database that supports triggers could be used. In the client, we use the Java-based Hypersonic SQL database engine to store the local data copies. Therefore, Mobisnap clients should run in any mobile device that supports Java 2. Until now, we have tested the clients in PCs running Windows and Linux and in Compaq iPAQ handheld computers running SavaJe OS. The Mobisnap middleware runs on top of the RDBMSs to implement the system specific functions, including, reservation management and mobile transaction processing (the database engines are used only to process SQL statements).

In our current prototype, the PL/SQL interpreters have several limitations, mainly the client interpreter that verifies if a transaction can be guaranteed. For example, although sub-queries can be processed, if they are used in a transaction, the transaction cannot be guaranteed. A complete implementation of an PL/SQL interpreter should allow to verify if any deterministic transaction can be guaranteed or not. Non-deterministic functions, such as querying the local time, can be handled by saving the value returned in the client and returning the same value in the server (similar to the *newid* function).

Other issues of the prototype are still being worked out, namely security, some reservation options such as reservation delegation between clients and improved caching management. In the future we intend to evaluate the im-

```
1    ------------- BUY 1 TICKET: train = "London-Paris 10:00"; day = '18-FEB-2002'; price <= 100.00 -------------
2    BEGIN
3      SELECT price, available INTO tkt_price, tkt_avail FROM trains
4                      WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002';
5      IF tkt_price <= 100.00 AND tkt_avail >= 1 THEN              -- checks seat availability
6        UPDATE trains SET available = tkt_avail - 1 WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002';
7        SELECT seat INTO tkt_seat FROM tickets
8                  WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002' AND used = FALSE;  -- get one available seat
9        UPDATE tickets SET used = TRUE, passenger = 'Mr. John Smith', price = tkt_price
10                WHERE train = 'London-Paris 10:00' AND day = '18-FEB-2002' AND seat = tkt_seat;
11       COMMIT (tkt_seat,tkt_price);                             -- commits and returns thicket information
12     ENDIF;
13     ROLLBACK;                                                 -- rollbacks when there is no seat available
14   END;
```

Figure 5: Mobile transaction reserving a new ticket in a train reservation system (declaration of variables is omitted).

id	type	table	column	condition	value	info
37-8	*escrow*	trains	available	train='London-Paris 10:00' AND day='18-FEB-2002'	2	≥ 0
37-9	*value-use*	trains	price	train='London-Paris 10:00' AND day='18-FEB-2002'	95.00	-
37-10	*value-change*	tickets	*	train='London-Paris 10:00' AND day='18-FEB-2002'	$(4A, false, \dots)$	-
37-11	*value-change*	tickets	*	train='London-Paris 10:00' AND day='18-FEB-2002'	$(4B, false, \dots)$	-

Figure 6: Reservations obtained in the train reservation system example: two seats are reserved.

Name	*small*	*large*
Duration (t)	12 hours	
Link failure interarrival mean	Exp(120 min)	
Link failure duration	Exp(36 min)	
Message latency	Exp(400 ms)	
Server failure interarrival mean	Exp(6 hours)	
Server failure duration	Exp(1 min)	
Number of clients	8	50
Initial stock ($stock_{init}$)	300	30000
Order quantity of each request (value of request)	round(0.5 + + Exp(1.5))	round(0.5 + + Exp(19.5))
Name	***good***	***bad***
Expected usage rate (e_u)	variable	
Prediction reliability ($sigma_{base}$)	0.04	0.64
Predicted vs. real conformance	$\approx 10\%$ in *small* $\approx 5\%$ in *large*	$\approx 55\%$

Table 2: Experimental parameters.

pact of reservations (and associated triggers) on the performance of the database server.

6 Evaluation

In this section we evaluate the effectiveness of reservations to support a mobile sales application through simulation. Due to space limitation we can only present a small part of the studied scenarios [17].

The mobile sales application maintains information about a set of products, including for each product, its current stock and price. A mobile salesperson uses a mobile device to submit requests from her customers. In the experiments presented in this paper, a single type of request is used: place a new order. Each new order is submitted as a mobile transaction identical to figure 1. To guarantee the result of these mobile transactions independently, mobile devices obtain reservations.

The experiments simulate the execution of the Mobisnap system, including the single server, a set of mobile clients and the network according to the parameters presented in table 2. Server failure parameters lead to 99.7% availability.

A network module simulates the communications be-

tween the server and the clients. We have modelled a simple mobile environment scenario where clients remain disconnected for long periods of time. End-to-end unavailability is 30%. We simulate end-to-end partitions using (client-server) link failures. However, we make no assumption on the cause (e.g. voluntary disconnection, energy restrictions, etc.) of each failure. Message latency has an exponential distribution to (roughly) model variable message delay. Latencies over 1 second are considered as message losses.

Our experiments model two deployment scenarios: a *small* and a *large* scale deployment of the mobile sales application (see table 2). The large scale scenario is used mainly to evaluate the influence of scale. We now explain the parameters for the *small* scale scenario.

In the *small* scenario, there are only eight salespersons (or mobile clients). For simplicity (and without loss of generality), we consider that there is just one product available, the initial stock is 300 and its price is 1. The quantity involved in each order is variable and based on an exponential distribution, as described in table 2 – this approach leads to generally small orders and rare large orders (overall average quantity is approximately 2). As the price of one instance of the product is 1, the value of each request is equal to the quantity in each order.

The requests are generated in mobile devices as follows. First, for each experiment, the expected usage rate, e_u, controls the expected value of all received requests, $exp_{total} : exp_{total} = e_u \times stock_{init}$. In our experiments, e_u varies from 55% to 175% modelling from weak to very strong demands.

Second, for each mobile client, we generate an individual predicted value of received requests, exp_i. exp_i is created randomly, such that $exp_{total} = \sum exp_i$. In practice, this value can be obtained from the history of clients using forecasting techniques [4].

id	type	table	column	condition	value	info
18-3	slot	datebook	*	day='17-FEB-2002' AND $hour \geq 8$ AND $hour \leq 13$	all records that satisfy the condition	-

Figure 7: Reservations obtained in the calendar example.

Third, we generate, based on the prediction reliability factor, $sigma_{base}$, the (expected) real value of requests received in each client, dem_i. dem_i is computed using a random variable with normal distribution (with $mean = exp_i$ and $sigma = sigma_{base} \times exp_i$). We evaluate two scenarios: one with good predictions (*good*) and one with bad predictions (*bad*). In table 2, we show, for each scenario, the average difference between the value of the requests created in our experiments and the predicted values (exp_i).

Finally, during each experiment, the submission of each request is controlled by a random variable with exponential distribution, as usual. The inter-arrival rate is equal to $t \times req_{avg}/dem_i$, with req_{avg} the average quantity of each request.

Our experiments evaluate two strategies to obtain reservations. In both, mobile devices obtain reservations that expire only in the end of the experiment. In the first, *stat X*, mobile devices obtain reservations only once, in the beginning of the experiment. *X* is the percentage of the stock the server reserves for itself (i.e., that clients cannot reserve). The remaining is reserved by mobile devices, proportionally to exp_i (in combination with value-use reservations). The second strategy, *dyn X*, extends the first by allowing mobile devices to request additional reservations when they cannot guarantee an order with the average order quantity. A request for additional reservations is serviced using the unreserved stock. To each user, the server concedes reservations proportional to the reservations he has obtained before (so that all mobile devices can get additional reservations).

In our experiments, reservations are always obtained from the server. This approach differs from other works [2], where multiple servers can be involved in the redistribution of escrowable resources. These approaches seem more appropriate for distributed settings where connectivity among servers is reliable and fast, and hosts do not have power restrictions that advise to disconnect them for some periods.

We have obtained the following results. First, the value of requests that can be *locally committed*, i.e., transactions that can be guaranteed independently in the mobile devices. Second, the value of requests that can be *immediately committed*, i.e., transactions that can be guaranteed either in the mobile device or synchronously contacting the server (if connectivity is available). As the maximum value (and percentage) of transactions that can succeed depends on the usage rate, we have also com-
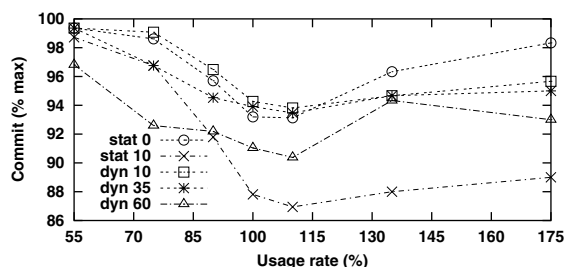

Figure 8: Locally committed transactions (good prediction).

puted the maximum value of transactions that could be committed in a system without failures. All results are presented as a percentage of this maximum value. Note that using the reevaluation mechanism, it is always possible to reach the maximum value of requests after the reservations expire.

Each experiment simulates a period of 12 hours. All results are the average of 10 runs. Different approaches are compared using the same request generation events.

6.1 Good prediction

In the first set of experiments, the difference between the actual and the expected demand is 5%, on average.

Figure 8 shows the transactions that can be locally committed. The results show that more than 85% of the maximum value of transactions can be guaranteed locally. As expected, the results are better as the usage rate deviates from 100%. For smaller usage rates, the excess of stock accommodates the unexpected requests. For bigger usage rates, as each mobile device could only obtain reservations for a fraction of the expected demand, exp_i, even if the actual demand is smaller than the expected one, all reservations tend to be consumed. Based on this rationale, we expected our results to get closer to 100% faster as usage rate deviates from 100%. Analyzing the experiments, we have discovered that the small scale of our example was introducing larger relative distortions than expected (results from the larger scale scenario confirm this conclusion – see figure 9).

Figure 10 shows the transactions that can be immediately committed. The results show that more than 95% of transactions can be immediately committed using reservations. The figure also show that a traditional client/server system (*clt/srv*) that tries to commit all transactions in the server immediately behaves much worse in this environment (where mobile devices do not have connectivity for large periods of time).
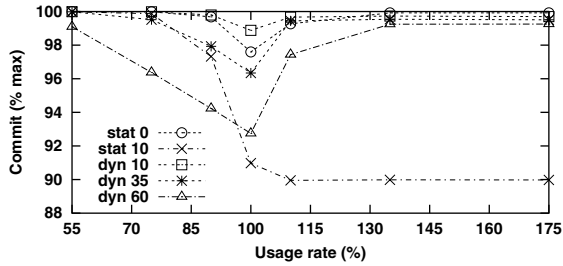
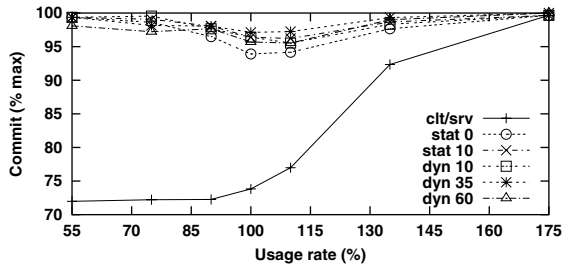Figure 9: Locally committed transactions (good prediction, large scale scenario).

Figure 11: Locally committed transactions (bad prediction).

Figure 10: Immediately committed transactions (good prediction).

Figure 12: Immediately committed transactions (bad prediction).

6.2 Bad prediction

In the second set of experiments, we investigate if reservations can also be used successfully when predictions are bad (the difference between the actual and the predicted demand is 55%, on average).

Figure 11 shows the transactions that can be locally committed. As expected, the results are worse than the obtained when the prediction is good. In this case, obtaining reservations dynamically is much better than obtaining reservations just once. For example, in the *dyn 60* scenario, the system can commit locally more than 85% of the maximum transactions that can be committed. However, dynamic strategies requires that mobile devices communicate with the server. These communication costs increase with the usage rate (until it is close to 110%) and with the increase of the stock reserved by the server [17]. However, these costs are small – e.g. in the *dyn 60* (resp. *dyn 30*) scenario with the usage rate of 110% (resp. 90%), clients contact the server once for each 6.25 (resp. 13.6) requests locally committed (in large scale scenarios, these values are much bigger).

Figure 12 shows the transactions that can be immediately committed. The values for the dynamic strategy are disappointing as they slightly improve the results of transactions locally committed. Analyzing the experiments, we found out that the problem was due to the reservations obtained initially. To solve this problem, we have used our dynamic strategy without obtaining any reservation initially, i.e., ignoring the predicted demand (this situation is equivalent to the unavailability of predictions).
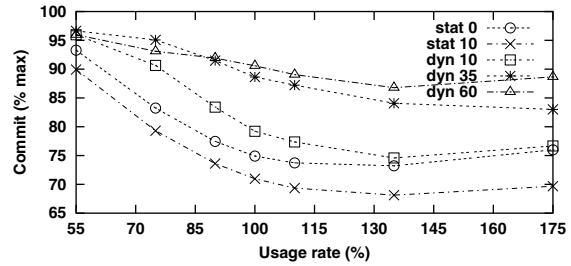
The results obtained were the following.

Figure 13 shows that more than 80% of the maximum transactions can be locally committed. As the system adapts to demand dynamically, it is impossible to locally commit some of the early transactions. This explains the increase of transactions committed locally with the increase of the usage rate – when there are more transactions, the influence of these early steps tends to be smaller (results from the large scale scenario, figure 14, also corroborate this hypothesis and show a very good adaptation of the dynamic approach).

Figure 15 shows the transactions that can be immediately committed. As expected, these results are much better than those obtained when clients obtain reservations initially. In this case, more than 95% of the maximum number of transactions can be committed immediately.

The results presented show that reservations can support a mobile sales application when it is possible to estimate the demand and even when such estimation is unknown.

7 Related work

Mobile data management has been addressed in several research projects and some solutions have even been integrated in commercial products. Some of the proposed approaches are presented in [1, 16, 19].

In Oracle Lite [14], mobile clients cache database snapshots. Transactions executed in clients are integrated in the master database using the new/old write sets and detecting write/write, uniqueness and delete conflicts. Conflict resolution rules can be associated with the database
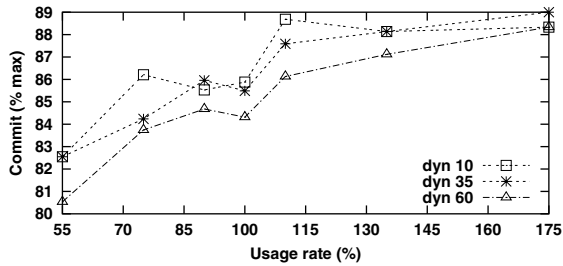
Figure 13: Locally committed transactions (bad prediction, unknown estimation).
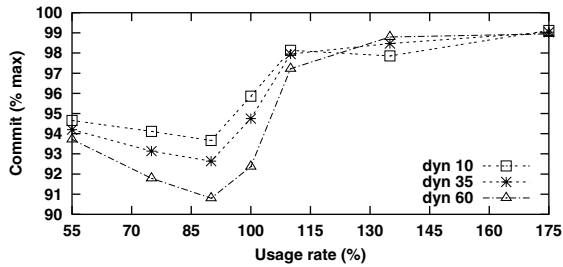

Figure 14: Locally committed transactions (bad prediction, unknown estimation, large scale scenario).

tables (and table fields). This state-based conflict resolution approach (as others [11]) is limited because the semantics of updates has been lost and it is impossible to specify any transaction-specific conflict resolution rule.

In the two-tier replication model [6], mobile nodes may propose tentative update transactions. These transactions are reapplied to the object master copy, verifying its validity using a specified acceptance rule. Invalid transactions are aborted and diagnostic messages are returned to the mobile nodes. In Bayou [20], data is replicated in a group of servers that synchronize epidemically. A primary server sets the commit order. Bayou updates allow generic conflict detection and resolution using dependency checks and merge procedures. The Mobisnap mobile transactions can be seen as an PL/SQL implementation of the updates proposed in these systems. However, unlike these systems, Mobisnap integrates mobile transactions with the reservation model to guarantee the results of transactions in the mobile devices.
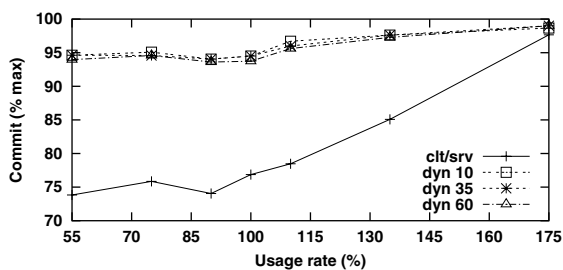

Figure 15: Immediately committed transactions (bad prediction, unknown estimation, small scale scenario).

IceCube [8] presents a reconciliation engine that tries to create an optimal single schedule, combining the maximum number of tentative actions executed in mobile devices – some action may be discarded to allow a larger set of actions to be executed. The IceCube reconciliation approach could be used in Mobisnap during the reintegration of mobile transactions received from the server and during the re-execution of transactions. In the multiversion reconciliation model [15], the server maintains multiple versions of the database history. The system tries to serialize each client transaction into one of these versions using a conflict resolution and a cost function specified with each transaction. Exploiting multiple data versions may lead to better results than the simple execution of mobile transactions in Mobisnap. However, it also imposes additional complexity in the server. Unlike Mobisnap, these systems do not include any mechanism to guarantee the results of transactions in mobile devices.

The escrow model [12] can be used to guarantee the result of some transactions in mobile devices [9]. The basic idea is to divide the available items of a commodity among several mobile sites – transactions that only use local items can be independently guaranteed. As explained, the Mobisnap escrow reservations implement and adapt this idea to an SQL-based system where reservations are used transparently. Therefore, any transaction can explore all reservations instead of only those it has been designed to. Moreover, the examples of section 4.5 show that additional reservations are necessary to guarantee the results of most transactions.

The use of escrow techniques can be generalized by exploiting object semantics [21]. The idea is to split large and complex objects into smaller fragments with certain constraints. If these constraints are honored, the modified fragments can be later merged using the semantic information available. This approach can be used only with some data types and it is more appropriate to object oriented databases.

TACT [22] defines a framework to control the divergence among several replicas integrating several metrics previously proposed. Although this approach can be used to increase the likelihood of reintegration success, it cannot guarantee the results locally. TACT demands applications to inform the system how each update affects the existent logical consistency units. This approach forces updates to know consistency units and it is error-prone, as any incorrect adjustment in any update leads to invalid divergence values. We believe that an automatic and transparent approach, as used in Mobisnap to check if mobile transactions can be guaranteed, is preferable.

A preliminary and incomplete version of Mobisnap was presented elsewhere [18]. Since then, important modifications have been introduced in the system, such as, the

introduction of new reservations and the modification of the transaction processing.

8 Final remarks

The Mobisnap database middleware system is designed to support applications that run on mobile computing environments. It supports independent operation combining mobile transactions and reservations.

A mobile transaction is a small PL/SQL program submitted by an application to modify the database state. As the final result of a mobile transaction is obtained running its program in the server, it can include conflict detection and resolution rules that exploit the semantic information associated with the operations.

A reservation provides some promise upon the database state, thus guaranteeing that no conflict will arise when a mobile transaction is executed in the server. Therefore, if a mobile client holds enough reservations, it can guarantee the final result of a mobile transaction independently.

Besides implementing this conflict avoidance mechanism in a middleware SQL-based system that allows legacy clients to continue to access the database, our reservation model presents the following new contributions. First, it includes several types of reservations. The examples presented in section 4.5 show that this feature is necessary to guarantee the result of most transactions. Second, the client transparently verifies if it can guarantee the result of any mobile transaction. Besides allowing mobile transactions to be written as usual PL/SQL programs, this property allows any mobile transaction to explore all available reservations. Finally, it is integrated with mobile transactions that allow the definition of conflict resolution rules. To our knowledge, this integration makes our system unique.

This paper has focused on the description of the Mobisnap reservation model and its integration with mobile transactions. The examples presented throughout the paper exemplify the use of the model with realistic applications. The system design and implementation described demonstrate the feasibility of the model. The evaluation of the reservation model shows that reservations can be used to support independent operation in a mobile sales applications even when it is impossible to estimate the expected demand. In the future, we expect to study the use of reservations in different settings and to evaluate the impact of reservations (triggers) in the performance of the database server.

More information on the Mobisnap system can be obtained from http://asc.di.fct.unl.pt/mobisnap.

Acknowledgements

References

[1] BARBARÁ, D. Mobile computing and databases - a survey. *Knowledge and Data Engineering 11*, 1 (1999), 108–117.

[2] CETINTEMEL, U., ÖZDEN, B., FRANKLIN, M., AND SILBERSCHATZ, A. Design and evaluation of redistribution strategies for wide-area commodity distribution. In *Proc. of the 21st International Conference on Distributed Computing Systems* (Apr. 2001), pp. 154–164.

[3] DAR, S., FRANKLIN, M. J., JÓNSSON, B., SRIVASTAVA, D., AND TAN, M. Semantic data caching and replacement. In *Proc. VLDB'96* (Sept. 1996), pp. 330–341.

[4] FRANK, T., AND VORNBERGER, M. Sales forecasting using neural networks. In *Proc. ICNN'97* (1997), vol. 4, pp. 2125–2128.

[5] GRAY, C., AND CHERITON, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the 12th ACM Symposium on Operating systems principles* (1989), pp. 202–210.

[6] GRAY, J., HELLAND, P., O'NEIL, P., AND SHASHA, D. The dangers of replication and a solution. In *Proc. of the 1996 ACM SIGMOD international conference on Management of data* (1996), pp. 173–182.

[7] GRAY, J., AND REUTER, A. *Transction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[8] KERMARREC, A.-M., ROWSTRON, A., SHAPIRO, M., AND DRUSCHEL, P. The icecube approach to the reconciliation of divergent replicas. In *Proc. of the 20th ACM Symposium on Principles of Distributed Computing* (2001), pp. 210–218.

[9] KRISHNAKUMAR, N., AND JAIN, R. Escrow techniques for mobile sales and inventory applications. *Wireless Networks 3*, 3 (1997), 235–246.

[10] KUENNING, G. H., AND POPEK, G. J. Automated hoarding for mobile computers. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (1997), pp. 264–275.

[11] KUMAR, P., AND SATYANARAYANAN, M. Flexible and safe resolution of file conflicts. In *Proc. USENIX Winter Technical Conference* (New Orleans, LA, USA, Jan. 1995), pp. 95–106.

[12] O'NEIL, P. E. The escrow transactional method. *ACM Transactions on Database Systems (TODS) 11*, 4 (1986), 405–430.

[13] ORACLE. Pl/sql user's guide and reference - release 8.0, June 1997.

[14] ORACLE. Oracle8i lite replication guide - release 4.0, 1999.

[15] PHATAK, S., AND BADRINATH, B. R. Multiversion reconciliation for mobile databases. In *Proc. 15th Int. Conference on Data Engineering* (Mar. 1999), pp. 582–589.

[16] PITOURA, E., AND SAMARAS, G. *Data Management for Mobile Computing*, vol. 10. Kluwer Academic Publishers, 1998.

[17] PREGUIÇA, N. *Data Management for collaborative mobile computing (in portuguese)*. PhD thesis, Dep. Informática, FCT, Universidade Nova de Lisboa, 2003 (expected).

[18] PREGUIÇA, N., ET AL. Mobile transaction management in mobisnap. In *Proc. of ADBIS-DASFAA 2000* (2000), vol. 1884 of *Lecture Notes in Computer Science*, pp. 379–386.

[19] SAITO, Y., AND SHAPIRO, M. Replication: Optimistic approaches. Tech. Rep. HPL-2002-33, Hewlett-Packard Laboratories, Mar. 2002.

[20] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. Managing update conflicts in bayou, a weakly connected replicated storage system. In *Proc. of the 15th ACM Symposium on Operating systems principles* (1995), pp. 172–182.

[21] WALBORN, G. D., AND CHRYSANTHIS, P. K. Supporting semantics-based transaction processing in mobile database applications. In *Proc. Symposium on Reliable Distributed Systems* (1995), pp. 31–40.

[22] YU, H., AND VAHDAT, A. Design and evaluation of a continuous consistency model for replicated services. In *Proc 4th Symposium on Operating System Design and Implementation (OSDI 2000)* (Oct. 2000), pp. 305–318.