# Transaction Policies for Mobile Networks*

Nuno Santos, Luís Veiga, Paulo Ferreira
INESC-ID/IST
Distributed Systems Group
Rua Alves Redol N 9, 1000-029 Lisboa
[nuno.santos, luis.veiga, paulo.ferreira]@inesc-id.pt

## Abstract

*Advances in wireless technology and affordable info-appliances are making mobile computing a reality. Such appliances can both communicate with a fixed station or with other info-appliances in an ad-hoc manner. In both scenarios the transaction paradigm is vital to provide applications with consistent access to durable data.*

*However, current mobile transactional systems fail to provide the so much needed adaptability to the large set of usage scenarios and applications semantics (e.g. disconnected work, relaxed ACID properties, etc.).*

*We present a transactional object-based mobile system, called MobileTrans, that supports the definition and enforcement of transaction policies. Such policies are separated from the application code and specify transactions behavior: (1) how data is fetched, (2) how updates are performed, (3) the degrees of consistency and atomicity required. Transaction policies can be either declarative (e.g. XML) or programmatic (e.g. Java, C#).*

## 1. Introduction

Mobile and portable devices, such as PDAs and laptops, are growing in number, features and diversity, making mobile computing a reality. These devices are equipped with wireless interfaces that allow them to communicate within mobile networks with fixed stations or with other info-appliances in an ad-hoc manner. Mobile networks are characterized by the mobility and/or disconnection of one or more hosts. In ad-hoc networks, dynamics is higher since nodes interact in an arbitrary way, they are connected for limited amounts of time and their connections may fail when nodes enter and leave the network. Furthermore, due to its size, portable devices resources are still scarce (memory, processing, power, etc.).

*This work was partially funded by FCT/FEDER.

In such scenarios, data sharing both between portable devices and between them and fixed stations, is clearly needed. However, the data required by the applications is not often accessible. Thus, due to the instability of the network topology, the transaction paradigm is necessary to ensure that shared data is accessed in a consistent way in the presence of disconnections. In addition, transactions provide a high level interface that relieve the developers of having to deal with the complexities of the environment. Therefore, a transaction system is an important tool for developing applications for such networks.

Current mobile transactional systems [17, 15, 13] already provide some mechanisms for dealing specifically with mobile networks. One is the ability to handle disconnections of the mobile host. Care must be taken so that frequent disconnections are not handled as failures causing the abortion of transactions. The other mechanism adapted to mobile scenarios is the relaxation of one or more properties of the ACID model [10]. In fact, the classical ACID transactions are clearly too restrictive for such mobile scenarios.

However, current mobile transactional solutions still have some important drawbacks leading either to unnecessary aborts or to unnecessary reduced data availability:

- Typically, if a transaction is not able to commit, it is forced to abort. In most proposals, alternatives instead of aborting the transaction are not provided or the existing alternatives are neither flexible nor simple enough to program by application developers.
- The requirements of applications in terms of the transaction properties can be very different. Some applications may require strict atomicity, while others may admit to discard some changes and still commit the transaction successfully. Other applications may desire to abort the transaction immediately if some node is not reachable while others may desire to postpone the commitment of the transaction.
- Different degrees of control (e.g. in terms of consis-

Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04)
0-7695-2141-X/04 $ 20.00 © 2004 IEEE

tency) may be required by developers for each application. Some developers may not desire to configure every detail of the transaction behavior, while others may even want to react to the changes of the environment in realtime.

- Finally, it is complex to specify the transactions behavior because it is necessary to think about all the situations and problems that can occur. In addition, if we need to adjust the transaction properties, it is necessary to change the code of the transaction.

In short, there is a lack of available options to prevent a transaction from aborting and there is a lack of flexibility and adaptability in current mobile transactional systems. Thus, current mobile transactional systems fail to provide the so much needed adaptability to the large set of usage scenarios and applications. This adaptability is strongly dependent on the semantics of applications.

To solve the above mentioned problems we designed and implemented MobileTrans, a transactional object-based peer-to-peer mobile system that supports the definition and enforcement of mobile transaction policies. By means of a policy specification and a carefully designed platform, MobileTrans supports the flexibility and adaptability needed for mobile networks of info-appliances with a minimum programming effort.

Each node running MobileTrans is able to access objects provided by any other node. When serving objects to others, a node works as a server; when getting objects from a remote node, it works as a client. Objects may contain references to other objects on others nodes, forming distributed graphs of objects. Objects are only allowed to be accessed in the context of a transaction. Transactions are executed under a distributed optimistic concurrency control protocol ensuring serializable histories.

The transactions' behavior can be adapted in run-time to deal with the specific scenarios and applications needs. In particular, MobileTrans transaction policies focus on dealing with disconnection and on specifying the minimum consistency and atomicity properties of a transaction:

- *Consistency*: It is possible to specify consistency rules that allow the usage of outdated versions of objects if the corresponding remote node is not available. This can be done per object or per set of objects.

- *Fetching*: The policy describes if (and how many) objects must be pre-fetched before executing the transaction or if objects should be fetched on demand while the transaction executes.

- *Delegation*: When a transaction is about to be committed, it is possible to delegate the transaction, i.e. to transfer the commit responsibility to other node.

- *Atomicity*: The policy can specify if the transaction can commit even if it not all nodes involved in the transaction are reachable, i.e. if some changes can be dropped.

- *Caching*: When executing transactions, it is possible to store both the fetched objects and the committed objects by local transactions, in a node's cache. This feature is essential for providing access to data during disconnection.

- *Failure Handling*: The policy is also responsible for determining how to react when the specified conditions of consistency, fetching and atomicity do not hold (due to contingencies of the network). For instance, the policy may specify that the transaction should be suspended until some event occurs. The policy is also allowed to change the transaction configuration in runtime in order to handle failures accordingly.

To implement a transaction, in addition to its code (i.e. objects methods), the application developer must specify and provide a transaction policy. The specification of a policy consists simply on assigning a set of attributes that will determine the behavior of a transaction. It declares the conditions that a transaction must hold and the procedure to be executed in case those conditions can not be enforced. The same policy can be applied to several transactions.

MobileTrans provides a high degree of flexibility in two ways. First, for the application developer: the policy can be either declarative (e.g. XML) or programmatic (e.g. Java, C#); in the later case, the developer has a high degree of control and can react and change, in runtime, the specifications enforced by the policy. Second, for the platform developer: MobileTrans is designed to be extensible in order to allow the inclusion of other attributes to support features not yet predicted.

Thus, the main contribution of MobileTrans is its support for transactional awareness in the sense that a transaction behaves according to a policy previously specified so that it can adapt to applications and mobile scenarios.

This paper is organized as follows. In the next section we present an overview of MobileTrans focusing both on its architecture and on the transaction model. Section 3 exposes the transaction policy mechanism of MobileTrans. In Sections 4 and 5 we present the details of the current implementation and its evaluation, respectively. Section 6 compares our work with others and in Section 7 we draw some conclusions and present future directions.

## 2. System Overview

We consider both a network which can be simply made of mobile nodes with no other infrastructure, i.e. an ad-hoc

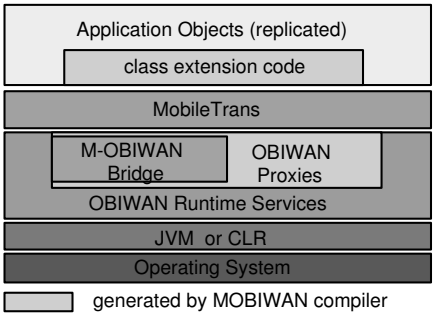


**Figure 1. System architecture overview.**

network, and also a scenario in which mobile nodes connect to the fixed network. Thus, mobile nodes may connect both to others mobile nodes or to other fixed nodes, typically for limited amounts of time, and their connections may fail due to its inherent mobility. In addition, due to its size, such portable devices are resource constrained (memory, processing, power, etc.).

While working as a server, a node provides information in the form of objects. Objects contain references to other objects building *graphs* of objects. A node where an object was created is called the object's *home node*.

Any object, which is part of a graph of objects, may be given a human readable name. Such objects can be seen as *roots* of a (sub)graph. An object graph may contain several named objects, i.e. roots. Applications obtain references to such objects, from a name service, given their name.

Nodes access objects (i.e. a single object, a full graph or subgraphs), provided by others, by replicating them locally; we call this operation, object *fetching*. Such access is done within a transaction and replication is performed automatically and transparently.

## 2.1. Architecture

MobileTrans is a middleware platform (see Figure 1) that provides support for the development and execution of applications. Applications are able to access objects according to a distributed transaction semantics. MobileTrans has two main components; its kernel, called MOBIWAN, and MobileTrans itself. We describe both in the next sections.

### 2.1.1 MOBIWAN

MOBIWAN is an evolution of OBIWAN [19, 8], a middleware platform that provides transparent, yet adaptive, incremental replication of object graphs. OBIWAN has been extended in order to both accommodate the transactional needs of MobileTrans and also to improve its performance on resource constrained devices (e.g. PDAs).

MOBIWAN supports the incremental replication of large object graphs into mobile nodes, allows the creation of dynamic clusters of data, and provides hooks for the application programmer to implement a set of application specific properties such as relaxed transactional support or updates dissemination (which MobileTrans uses).

MOBIWAN is a set of runtime services on top of either the Java or .Net virtual machines. It is comprised of five parts: runtime services, a mobile-device bridge (for communication purposes between nodes), an open-compiler (that generates code automatically), proxies and class extension code (automatically generated). The base runtime services include, mainly, object registration, name service, object repository discovery and connection, and custom event-handling.

For the purpose of this paper, the most relevant MOBIWAN data structures are the proxy-out/proxy-in pairs [18]. A proxy-out (an out-going proxy) stands in for an object that is not yet locally replicated. For each proxy-out there is a corresponding proxy-in (an in-coming proxy).

When a not yet locally replicated object is invoked for the first time, the corresponding proxy-out interacts with its counterpart proxy-in (residing at the remote node) to perform the replication of the corresponding object. This enables the incremental replication of object graphs. Once objects are locally replicated, invocations are direct, i.e. with no indirection at all. This is achieved by careful combination of proxies and referring objects. Proxies also mediate object updating, i.e. when local replicas are sent back to remote nodes. This happens when a transaction commits.

Replication of more than a single object is also permitted, obviously. In particular, the programmer may specify at run-time the amount of objects that should be replicated. So, a whole cluster (i.e. a subgraph of objects) can be replicated in a single step instead of replicating a single object individually. This mechanism is extremely useful as it allows to replicate, as a whole, a set of objects that are to be accessed within a transaction.

Communication among nodes is performed using a bridge based on a set of web-services. These services are set-up on each node, and are invoked by proxies-out and runtime services. The web-services encapsulate all communication and delegate requests on to other nodes.

An open-compiler, called *obicomp*, automatically generates code for proxy classes and augment application classes. This augmentation process does not interfere with application-logic methods. It simply implements, automatically, special-purpose code so that classes are able to create replicas of their instance objects. Proxy and class-extension code do not include communication related code in order to increase flexibility in object replication and updating.

To simplify design, increase modularity and allow different, more sophisticated replication techniques, every

step of object replication (into a mobile node) and update (back to remote nodes) is performed by handling specially defined events. Specific events (e.g., *before-replica*, *after-replica*, *before-update*, etc.) are triggered by MOBIWAN. Default handlers for these events are implemented in the base runtime services performing basic semantics expected. Nonetheless, the application, either explicitly or in its declarative setup/configuration can define different handlers with added versatility, flexibility, different Qos and fault-tolerance, either replicating objects or performing their update. These primitives are used by MobileTrans as they are the basis upon transactional concurrency control and transactional policy mechanisms are built.

#### 2.1.2 MobileTrans

MobileTrans runs on top of MOBIWAN and makes extensive use of the provided hooks. In other words, it implements the event handlers related to object replication (i.e. fetching of objects being accessed within a transaction) and to object update back into the remote nodes (i.e. when the transaction commits). These event handlers implement specific transaction mechanisms concerning concurrency control, atomicity, etc.; in particular, they behave according to a *transaction* policy previously defined.

### 2.2. Transaction Model

Within a transaction, object graphs provided by other nodes (the *coordinator* and the *cohort* nodes in transactional terminology) are replicated (i.e. fetched) into a mobile node where they are then read and written. The set of fetched objects that are read/written is called the *readset*/*writeset*; the union of the read and the write sets is the *dataset*. A transaction that performs reads and writes is called a *read-write* transaction; a transaction that only performs reads is a *read only* transaction.

Concurrency control is based on a distributed multiversion parallel validation algorithm (MVPV) [2]. MVPV is an optimistic concurrency control protocol that ensures serializable histories. It consists on three phases: the *read*, the *validation* and the *write* phases. MVPV is also a multiversion scheme, where new object versions are created once updates are made. Each running transaction, when in the read phase, is provided with a consistent view, which is the result of a serial execution of transactions that has already committed when the transaction started. The distribution of this algorithm is based on the 2PC protocol.

MVPV has important benefits for a mobile environment. Since a transaction does not see the results of concurrent read-write transactions, transactions behave predictably and read-only transactions need not be validated. It is only necessary to validate against read-write transactions to enforce

```
void AddAppointment(Appointment ap, String[] hosts,
    TransactionPolicy policy) {
    Transaction t = new Transaction(policy);
    t.Begin();
    for (int i = 0; i < hosts.length; i++) {
        bool unschedulled = true;
        Appointment a = t.Get("Schedule",hosts[i]);
        for(Appointment i=a; a!=null; a=a.next()) {
            if (a.Collide(ap)) {
                unschedulled = false; break;
            }
        }
        if (unschedulled && a != null) { a.Append(ap); }
    }
    t.Commit();
}
```

**Figure 2. Example of a transaction.**

serializable histories. Furthermore, cascading rollbacks can not occur; the cohorts do not store any state for remote transactions; and there is no need to contact the cohorts if the transaction is aborted. However, since a new version is created once an object is updated, the amount of storage space need is significant. However, this is not a drawback because such versions are only kept if there is enough space (subject to policy specification).

MobileTrans uses a modified version of the MVPV algorithm, in two ways. First, it is adapted for handling objects instead of database relations. Second, based on policy specification, it provides configuration facilities adapting the transaction behavior and increasing its flexibility.

Application developers must use a set of primitives to implement a transaction: Begin, Get, Set, Commit and Abort. A transaction's lifespan can be divided in two phases: the *fetch phase* and the *commit phase*. The fetch phase consists on the read phase of the MVPV. It starts with a Begin primitive and admits arbitrary invocations of the Get and Set primitives for fetching and updating objects, respectively. The commit phase begins when Commit is performed, and the transaction is aborted with Abort. Commit phase performs both the validation and the write phases of MVPV.

## 3. Transaction Policies

Consider a scenario where each network node exports a schedule which contains a list of appointments. It is a linked list of objects, each representing an appointment. The schedule head list is assigned a well know name Schedule. The code of Figure 2 presents a trivial yet motivating example of a transaction in such scenario.

This transaction consults the lists of a set of nodes and attempts to schedule a new appointment. For example, if this method was called with hosts={"a.pt","b.pt","c.pt"}, the transaction would fetch the schedules of the listed hosts. For each schedule, the transaction checks if the new appointment conflicts with an already scheduled appointment. If

| Attributes | | | | |
|---|---|---|---|---|
| **Name** | | **Value** | | **Arguments** |
| consistency | .object | *required* | | – |
| | | *replica* | | – |
| | | *dispensable* | | – |
| | .degree | *high* | | – |
| | | *medium* | | – |
| | | *low* | | – |
| fetching | .object | *random* | | depth |
| | | *node* | | depth, node |
| | | *randset* | | depth,{node} |
| | .mode | *ondemand* | | – |
| | | *prefetch* | | {obj} |
| delegation | .coordinator | *random* | | – |
| | | *node* | | node |
| | | *randset* | | {node} |
| | .responsibility | *local* | | – |
| | | *foreign* | | – |
| atomicity | .object | *mandatory* | | – |
| | | *tentative* | | – |
| | .degree | *high* | | – |
| | | *low* | | – |
| caching | .read | *yes* | | – |
| | | *no* | | – |
| | .write | *yes* | | – |
| | | *no* | | – |
| failure | .consistency .fetching .delegation .atomicity .user.* | *abort* | | – |
| | | *retry* | | attribute |
| | | *suspend* | *.timeout* | time, attribute |
| | | | *.reshape* | attribute |
| | | | *.user* | attribute |

**Table 1. Attributes of transactions.**

it does, the appointment is not added to the schedule. The transaction does not demand every schedule to have a free slot. This example is used for the remainder of the paper to help understand how MobileTrans can support different behaviors for the same transaction, without changing its code.

## 3.1. Configuration Facilities

In MobileTrans, it is possible to specify the exact behavior of a transaction according to a set of predefined parameters called *attributes*. The full list of MobileTrans transaction attributes is presented in Table 1.

An attribute is identified by a unique *name*. This name is hierarchical (a sequence of identifiers separated by dots) to provide a better organization of the attribute namespace. This structure is purely syntactic. For example, consistency.object and consistency.degree are names of attributes.

An attribute can be assigned a *value*. Some values, such as *suspend.timeout*, can also be parameterized with arguments. The set of values that can be assigned to an attribute is called the *domain*. Each attribute has its own domain. The domain of attribute consistency.object is the set {*required*, *replica*, *dispensable*}. The value namespace is also organized hierarchically (e.g. *suspend.timeout* and *suspend.reshape*).

It is said that an attribute is *instantiated* when a value is assigned. The *instance* is the pair $i_{attr} =< name, value >$, which represents that assignment. Instances are valid in the context of a transaction and are only valid during the transaction life span. Instances of one transaction do not interfere with other transactions.

Some attributes are relevant for the whole transaction, while others refer to a specific object of the transaction's dataset. The former ones are called *transaction attributes*, the later ones are called *object attributes*.

A transaction is said to be *configured* when all its attributes are instanciated. This means that MobileTrans is now fully instructed about the desired behavior for that transaction and the transaction can be executed. Thus, configuring the behavior of a transaction consists on choosing the values of the transaction's attributes.

In the remainder of this section, we present the major configuration facilities of MobileTrans and describe the attributes relevant for each facility.

### 3.1.1 Consistency

MobileTrans provides facilities to control the quality of the fetched data, by specifying the relevance of each object's consistency to a transaction. There are two relevant attributes. One is the object attribute consistency.object. It is evaluated by MobileTrans whenever the object is fetched. The other is the transaction attribute consistency.degree. It defines degrees of consistency for the transaction according to the consistency requirements of objects. This attribute is read before the read stage begins and is used every time an object is fetched.

The semantics is as follows. If the value of consistency.degree is *high*, it is required that all objects be fetched from its home nodes. If some object is required for reading, the transaction can not proceed until it is possible to get a replica of the object from the home node. This ensures that the transaction gets a consistent view of data.

If the consistency.degree value is *medium*, only the home nodes of the objects marked as *required* must be directly reachable. If the home nodes of objects marked as *replica* or *dispensable* are not reachable, MobileTrans must provide a copy of the objects from the local cache or from caches of online nodes. Is not guaranteed that objects fetched from caches are consistent.

Finally, if the degree is *low*, the home nodes of the data marked as *required* must be reachable, and, at least, some version of the objects marked with *replica* must also be reachable. If the data marked as *dispensable* is not reachable, the transaction may still proceed and are returned null references to the transaction.

In the example, suppose the schedule of a is marked as *required*, b as *replica* and c as *dispensable*, respectively. This makes sense, if hosts a and b are more relevant than c. If all

nodes are online, it makes sense that the degree be *high*, and the transaction can be executed among all the nodes. However, if c is not online, it may still be important to execute the transaction. Hence the degree would be set to *low*. The intermediate attributes can also be useful if it is preferable to fetch inconsistent object versions than none at all.

#### 3.1.2 Fetching

In MobileTrans it is possible to fetch an object according to two semantics: on a *prefetch* basis, which means that objects (sub)graphs are fetched into the transaction's context before the transaction starts executing; or *on-demand*, in which case, objects (sub)graphs are only fetched upon the first access, on transaction execution.

For this purpose, the object attribute fetching.object must be provided. It informs MobileTrans about the host from which to fetch the object and the depth[1] of its graph. There are three possibilities: (1) to fetch the object randomly from one of the nodes currently online (*random*); (2) to fetch randomly from a list of possible nodes (*randset*), where it is necessary to provide the name of the possible hosts; or (3) to fetch the object from a specific node (*node*). This attribute is evaluated before each object is fetched.

The fetching.mode attribute tells MobileTrans if it is necessary to prefetch the objects or not. In the first case, the value is *prefetch*, and it is necessary to provide the identifier of the root object whose graph is to be prefetched. Objects not listed are fetched on demand. If the value is *ondemand*, every object is fetched on demand, i.e. when read by the first time.

In the schedule example, suppose that host c is about to leave the network or to become powerless. In this case, it is interesting to prefetch the whole object graph to prevent the inaccessibility of data. Since the other nodes will continue online, objects can be fetched on-demand. But, if b leaves the network during the transaction, by specifying the *fetching.object* attribute, MobileTrans, can be instructed to fetch a version of that object from other node.

#### 3.1.3 Delegation

The coordinator is responsible for the initiation and coordination of a transaction. In MobileTrans it is possible to delegate the responsibility of commitment to another node, i.e. the role of "coordinator" will be performed by other node. This role consists on the execution of the 2PC protocol.

This feature is configured by assigning the transaction attribute delegation.responsibility with values *local* or *foreign*. The former means that the coordinator remains the same. The later activates delegation.

[1]Depth of a root object, is the number of object references that need to be transversed from the root to any other object, reachable from the root.

If delegation is specified, it is necessary to assign the transaction attribute delegation.coordinator which identifies the node of the new coordinator. Similarly to fetching, there are three possibilities of identifying the nodes: *random*, *ranset* or *node*.

#### 3.1.4 Atomicity

MobileTrans can be instructed to drop some changes made by the transaction. It is possible to specify the identification of the objects that must be updated during commit and the objects that do not cause the transaction to abort in case it is not possible to commit them successfully (e.g. the home node is not online).

For this purpose it uses two attributes: the object attribute atomicity.object, which is evaluated by MobileTrans whenever the commit protocol is executing; and the atomicity.degree attribute that defines the desired atomicity degree for the transaction. The latter attribute is necessary before the commit protocol starts.

The procedure is as follows. If atomicity.degree is *high*, the home nodes of all the modified objects (writeset) must be reachable and the local commit of all the participants must be valid. All the changes must be stored, thus providing the higher level of atomicity.

The alternative is to set atomicity degree as *low*. In this case, only the home nodes of the objects marked as *mandatory* have to commit successfully. If the home nodes of objects marked as *tentative* are not reachable or if they cannot commit locally, these changes can be discarded and the transaction can still commit.

In the example, suppose that the schedules of a.pt and b.pt were marked as *mandatory* and c.pt as *tentative*. If the degree was *high*, the transaction would only commit if all changes where submitted. But if the degree was dropped to *low*, if host c.pt becomes unreachable during commit, for example, the transaction would still commit. Obviously this assignment is dependent on the application semantics.

#### 3.1.5 Caching

MobileTrans can be configured to store copies of objects that belong to the transaction's dataset in the local cache. Two attributes are relevant. The caching.read attribute, informs MobileTrans if the fetched replica of object must be locally cached or not. The caching.write attribute, if the object is modified and is successfully committed, tells MobileTrans that its new version must be stored (or not) in the local cache. By specifying such attributes, it is possible to make data available even if it is not possible to fetch objects from its home nodes. Thus, this facility can be quite useful to keep on working even while disconnected.

### 3.1.6 Failures

If the conditions specified by the current attribute instances can not hold, MobileTrans provides a facility for describing how such failures should be handled. The relevant failure conditions are associated with the attributes failure.consistency, failure.delegation, failure.atomicity and failure.fetching. The first two are evaluated when the degrees of consistency or atomicity, respectively, can not be ensured. The others are evaluated when fetching can not be executed.

The value assigned to each of these attributes informs MobileTrans of the action that must be taken to overcome the respective failure. However, given the set of available actions, application developers may need to execute a recovery procedure that involves a sequence of actions. Therefore, MobileTrans provides a way for describing such sequence. It consists on allowing developers to add new attributes to the failure.user.* namespace which represent a step in the recovery procedure. These attributes can be used as arguments of the value associated with the failure attributes. Thus, MobileTrans can be informed of the next attribute to be read and, hence, to describe the next action to be taken.

All the failure.* attributes have the same domain. When the *abort* value is assigned, it means that the transaction should be immediately aborted. The other values are designed to make possible to recover the transaction by reevaluating the condition that gave rise to the failure. The *retry* value, performs this reevaluation and it must be given, as argument, the user attribute that must be read in case the associated condition can not still be performed. Since this action performs a new attempt immediately, MobileTrans also provides the *suspend.** values which means that the transaction should be suspended until some specific event happens. Currently, there are three predicted events, which define three attributes: *suspend.timeout*, suspending the transaction for a specified period of time; *suspend.reshape*, which suspends the transaction until the system detects a change of the network topology (some node enters or leaves the network); and *suspend.user*, where the transaction is suspended until it is sent an event by the application (dependent on the application semantics) using the MobileTrans API. All the *suspend.** values expect, as argument, a user attribute. When the suspend event is triggered, MobileTrans consults the user attribute which contains the next action to be performed.

For example, suppose that when some object could not be fetched, we would like to retry the fetching after the network topology has changed, to check if the searched node is in range. If the node does not appear, the transaction must abort. In this case it would be necessary to define the following instances:

$i_1$ =<failure.fetching,*suspend.reshape(*failure.user.attr1*)*>,

$i_2$ =<failure.user.attr1,*retry(*failure.user.attr2*)*>,

$i_3$ =<failure.user.attr2,*abort*>.

```
POLICY(AName n, CtxData c, EnvData e) → AValue {
    AppData a;
    switch(n) {
        case attribute_1.name:
            // value_1x ∈ attribute_1.domain
            if rule_11(c,e,a) → <value_11, arg_11(c,e,a)>;
            ...
            if rule_1n(c,e,a) → <value_1n, arg_1n(c,e,a)>;
            if TRUE → <value_1−default>;
        ...
        case attribute_m.name:
            // value_my ∈ attribute_m.domain
            if rule_m1(c,e,a) → <value_m1, arg_11(c,e,a)>;
            ...
            if rule_mn(c,e,a) → <value_mn, arg_11(c,e,a)>;
            if TRUE → <value_m−default>;
    }
}
```

**Figure 3. Model of the transaction policy.**

Policies make possible that during these recovery procedures, any attribute of the transaction can be reassigned. Thus, it is possible to describe complex recovery procedures according to the application semantics.

## 3.2. Policy Rationale

The specification of the attribute instances for a transaction in MobileTrans is a *transaction policy*. Until now, we have assumed that the transaction is fully configured before the transaction begins. However, using a transaction policy, the attribute instances can be provided at runtime. When necessary, MobileTrans asks the transaction policy for the value of each attribute. The policy returns a value from the attribute domain along with its arguments.

The model of the transaction policy is depicted in Figure 3. A transaction policy can be viewed as a sequence of *rules*. A rule is a function which associates an attribute to a value of the attribute's domain, according to some test condition. These conditions can take into account data items from several sources:

- *Context data*: This information is provided by MobileTrans in the CtxData data structure. This data structure contains information related to the transaction (e.g. the read set, the write set, the transaction identifier). In case the requested attribute is an object attribute, this parameter also contains information about the object, such as: the object's name and the object's home node name.

- *Environment data*: It is also provided by MobileTrans to the policy in the EnvData data structure. It is a resource to get information about the environment. Namely, it describes: the current online nodes, the available bandwidth and the current power availability.

```
<!— Excerpt of conf.xml —>
<transaction>
    <attribute name="consistency.object">
        <rule cond="o[host]=='a.pt'" value="required"/>
        <rule cond="o[host]=='b.pt'" value="replica"/>
        <rule cond="o[host]=='c.pt'" value="dispensable"/>
        <rule cond="true" value="required"/>
    </attribute>
    <attribute name="consistency.degree">
        <rule cond="true" value="high">
    </attribute>
    ...
</transaction>

// Application code
TransactionPolicy p = new TransactionPolicy("conf.xml");
...
```

**Figure 4. Declarative specification.**

```
class A {
    AValue RuleCO(AName n, CtxData c, EnvData e) {
        if (c["object.host"] == "a.pt") {
            return new AValue("required");
        }
        if (c["object.host"] == "b.pt") {
            return new AValue("replica");
        }
        if (c["object.host"] == "c.pt") {
            return new AValue("dispensable");
        }
        return new AValue("required");
    }
    AValue RuleCD(AName n, CtxData c, EnvData e) {
        return new AValue("high");
    }
}
// Application code
TransactionPolicy p = new TransactionPolicy();
A a = new A();
p["consistency.object"].AddRule(a.RuleCO);
p["consistency.degree"].AddRule(a.RuleCD);
```

**Figure 5. Programmatic specification.**

- *Application data*: The policy can also contain internal semantic information related to the application. Thus, the application can also be determinant in the way values are assigned to attributes. This data is represented as the type AppData.

Several rules can be associated for each attribute. When the policy is queried, the rules of the corresponding attribute are evaluated in cascade. MobileTrans requires that each attribute provides a default rule.

### 3.3. Policy Specification

A policy is implemented as a class which implements the policy rules. The application developer is responsible for creating and initializing the policy that will be used for configuring the transaction. There are two approaches for such initialization: *declarative* and *programmatic*.

A declarative specification consists on describing the policy rules on a XML file. The initialization of the transaction policy consists on loading the policy with rules stored in the file. An example can be seen in Figure 4.

The programmatic approach (see Figure 5) consists on providing the transaction policy all the required rules. The rule is a method that receives, as arguments, the attribute name, the context data and the environment data, and must return a value. Its implementation fully depends on the application. These methods are called by MobileTrans when it is necessary to get the values of attributes.

## 4. Implementation

The prototype implementation was developed both on .Net and .Net Compact Frameworks, using C# as primary language. The OBIWAN runtime services and MobileTrans were developed using Remoting services. The MOBIWAN Bridge was developed as a web-service, and runs on top of Internet Information Server. The obicomp compiler automatically generates proxies coded in C#. Parsing of class code only accepts C# source code and extends classes with replication-specific code. Due to this last feature, applications must still be developed only in C#. This is not a major drawback. Nonetheless, we intend to address this issue by also parsing VB.Net code. The platform library and proxy code need not be changed since the .Net VM is able to mix execution of assemblies written in different languages. In application code, the programmer simply needs to insert instructions to discover and fetch repositories. From that on, only application-logic code is required and communication is transparently handled by proxies. Once objects are replicated, their local proxies are discarded by the local garbage collector. The programmer never needs to invoke object replication explicitly.

## 5. Evaluation

To demonstrate how transaction policies can be used to improve the adaptability of transactions, consider the scenario referred in Section 3. Suppose that nodes a, b, c and x are in a room. Then, x decides to execute the transaction AddApointment, with the other nodes names as argument.

Before the transaction begins, suppose that x already knows that a is going to leave the room temporarily but will come back again. So, it prefetches a's schedule while it is online. The transaction executes normally on x. If the schedule was not changed, the transaction is read-only. Therefore, the transaction can commit even if a does not come back online. Otherwise, if it is a read-write transaction, the transaction only commits if a comes back online.

Suppose that it is a read-write transaction, but a had to go out definitely. In this case the user may not desire to

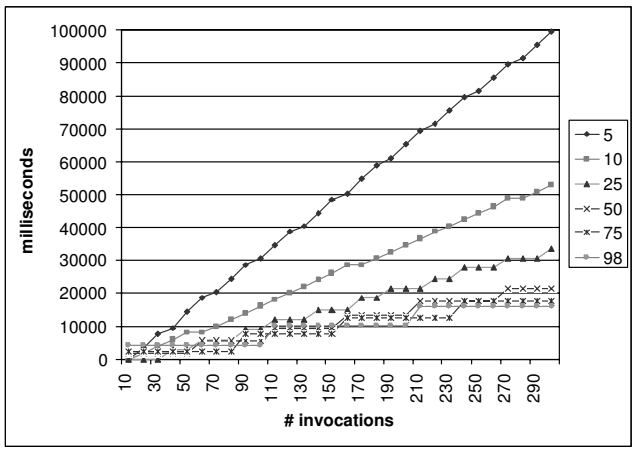


**Figure 6. Performance with various replication depths.**

wait indefinitely. He may prefer to discard the updates on a, but still commit the transaction with the other online nodes b and c. Thus, the atomicity degree of the policy can be dropped to allow the commitment of the transaction.

Consider instead that a must get the updates and the transaction must commit, but node x has to leave the room. Nodes b and c still remain in the room. In that case, the commitment of the transaction can be delegated to one of the nodes. Thus, when a returns, all the transaction cohorts are present and the transaction will be able to commit.

In these situations, just by specifying the transaction policy, without changing the transaction code, the transaction is able to adapt according to the application needs.

### 5.1. Quantitative Evaluation

Besides the qualitative evaluation, we analyzed the prototype performance with a micro-benchmark: series of iterations were executed on a hypothetical list of appointments with 1000 elements with a payload of 64 bytes each.

The performance tests were executed with the following infrastructure: a Pentium 4, 2.8 Ghz, 512 MB PC, an IPAQ 3360 Pocket PC and a Bluetooth USB adapter.

Cold connection setup time was about 8500 ms in each experiment. The replication mechanism was configured, by means of different policies, to replicate objects on-demand with a depth of 5, 10, 25, 50, 75 and 98 objects at a time. This way, every time a proxy is replaced and the corresponding object is replicated, a number of others, referenced by it, are also pre-fetched.

The limit depth, 98, is imposed by stack restriction on .Net CF. The graph shows that replication performance is latency-bound, i.e., it is most efficient when several (more than 25) objects are replicated each time.

These are rather encouraging result for various reasons: i) naturally, on-demand object replication of objects masks communication latency and minimizes memory usage by applications, ii) the number of objects pre-fetched for near optimal results needs not be too large (25 or 50). Best results are achieved with higher replication depths (75 or 98) but these could waste more memory if only a few of the objects pre-fetched are actually accessed.

## 6. Related Work

Most research on mobile transaction systems considers networks where mobile hosts (MHs) connect wirelessly to fixed base stations. In other words, they do not consider adhoc networks of MHs. Thus, their solutions are typically client-server based.

In such systems, one important issue is the disconnection of MHs. Concerning disconnection, the most common approaches (e.g. Pro-Motion [17], Clustering [16] and Gold Rush[4]) were inspired by the Coda file system [12]. When the MH is connected to the fixed network through a base station, data is cached in the MH. While the MH is disconnected, data locally cached can be accessed inside transactions and logged in the MH. Upon reconnection, transactions are reconciled in the database server (accessed through the base stations). The Prewrite model [15] handles disconnection differently, dividing the transaction in two phases: one that must be executed online, and other that can be executed while disconnected.

Some mobile transaction systems use semantic information to adapt the behavior of transactions. In Pro-Motion data is encapsulated in *compacts* which allow the definition of consistency rules to be applied to such data set as a whole. In Clustering it is possible to specify consistency degrees among replicated data. Moflex [13] also provides a mechanism for describing the associated behavior while crossing wireless cells. With Toggle [7], it is possible to specify different atomicity and isolation degrees, by dividing a transaction in vital and non-vital subtransactions.

It's worthy to note that most of the above mentioned proposals use semantic atomicity as its correctness criteria. This has two drawbacks. First, it is not possible to ensure serializability, thus it is not general enough for developing transactions that require serialization. Second, the developer, for each transaction, must implement a compensating transaction, which can be complex and time consuming.

Research has been done also to extend transaction models (ETM) [3], in order to make them more suitable for the requirements of mobile applications. Some frameworks where developed with focus on the design of such ETMs using application's semantic information. The ACTA framework [5] constructs a theoretical model that helps reasoning about and compare different ETMs. Inspired by ACTA,

Asset [1] allows users to define custom transaction semantics for specific applications. It provides primitives that can be composed together to define a variety of ETMs. Aster [9]presents a formal method for the systematic synthesis of transactional middleware, based on the formal specification of transaction properties and stub code generation.

Solutions have also been proposed that provide facilities for configuring the concurrency control protocol according to application's semantic information. CORD [11] introduced the Concurrency Control Language (CCL); CCL allows the application developer to specify a concurrency control policy tailored to the behavior of the transaction manager. PJama [14] is a proposal that is inline to what is pursued by MobileTrans. Using PJama, application developers can define the desired transaction behavior while maintaining transaction independence.

In short, in all these proposals, the support provided for transactional policy specification is not appropriate for the usage scenarios considered by MobileTrans.

## 7. Conclusions and Future Work

In this paper we introduced the design and implementation of MobileTrans, a transactional object based system for mobile networks, that supports the definition and enforcement of transaction policies. MobileTrans uses a distributed optimistic concurrency control protocol.

Application developers, when using MobileTrans, along with the transaction code, must specify a transaction policy. This specification can be declarative by means of a configuration file or programmatic which allow a higher degree of control, even in runtime. Using transaction policies, developers are able to configure several aspects of the transaction behavior, such as the specification of consistency and atomicity requirements, how the objects are fetched, if data is to be locally cached, if delegation is required and how failures are handled. Thus, transaction policies provide a powerful and flexible mechanism for configuring the behavior of transactions according to application semantics.

MobileTrans is a platform under development, hence, several features are now under study (e.g. refining the concurrency control protocol, adding agent based technology, automatic determination of the optimal depth on fetching). Thus, it will be necessary to add new attributes in order to increase the application awareness.

We also intend to increase the expressiveness of how policies are declaratively specified. For that purpose, we are considering an approach similar to the one presented in [6], for developing a transaction policy specification language.

## References


[1] A. Biliris et al. ASSET: A system for supporting extended transactions. In R. T. Snodgrass and M. Winslett, editors, *Proc. of the 1994 ACM Intl. Conf. on Management of Data*, pages 44–54, 1994.

[2] D. Agrawal, A. Bernstein, P. Gupta, and S. Sengupta. Distributed multi-version optimistic concurrency control with reduced rollback. *Distributed Computing*, 2(1), 1987.

[3] N. S. Barghouti and G. E. Kaiser. Concurrency control in advanced database applications. *ACM Computing Surveys*, 23(3):269–317, 1991.

[4] M. Butrico, H. Chang, A. Cocchi, N. Cohen, D. Shea, and S. Smith. Gold Rush: Mobile transaction middleware with java-object replication. In *Proc. of the Third USENIX Conference on Object-Oriented Technologies*, pages 91–101, 1997.

[5] P. K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, 19(3):450–491, 1994.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The ponder policy specification language. *Lecture Notes in Computer Science*, 1995:18–39, 2001.

[7] R. A. Dirckze and L. Gruenwald. A toggle transaction management technique for mobile multidatabases. In *Proc. of the CIKM 98*, pages 371–377, 1998.

[8] P. Ferreira, L. Veiga, and C. Ribeiro. Obiwan - design and implementation of a middleware platform. *IEEE Trans. on Parallel and Distributed Systems*, 14(11):1086–1099, Nov 2003.

[9] G. S. Blair et al. The role of software architecture in constraining adaptation in component-based middleware platforms. In *IFIP/ACM Intl. Conf. on Dist. Systems Platforms*, pages 164–184. Springer-Verlag New York, Inc., 2000.

[10] J. N. Gray and A. Reuter. *Transaction Processing: Concepts*. Morgan Kaufmann, 1993.

[11] G. Heineman and G. Kaiser. The CORD approach to extensible concurrency control. In *13th IEEE Intl. Conf. on Data Engineering*, pages 562–571, April 1997.

[12] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[13] K. Ku and Y. Kim. Moflex transaction model for mobile heterogeneous multidatabase systems. In *Proc. of the 10th Intl. Workshop on Research Issues in Data Engineering*, 2000.

[14] L. Daynes et al. Customizable concurrency control for persistent java. In S. Jajodia & L. Kerschberg, editor, *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[15] S. K. Madria and B. Bhargava. A transaction model for improving data, availability in mobile computing. *Distributed and Parallel Databases: An International Journal*, 10(2):127–160, 2001.

[16] E. Pitoura and B. Bhargava. Maintaining consistency of data in mobile distributed environments. In *Proc. of 15th Intl. Conf. on Distributed Computing Systems*, 1995.

[17] K. Ramamritham and P. K. Chrysanthis. A taxonomy of correctness criterion in database applications. *Journal of Very Large Databases*, 4(1), 1996.

[18] M. Shapiro. Structure and encapsulation in distributed systems: the proxy principle. In *Proc. of the 6th Intl. Conf. on Dist. Systems*, pages 198–204, Boston, May 1986.

[19] L. Veiga and P. Ferreira. Incremental replication for mobility support in OBIWAN. In *The 22nd Intl. Conf. on Distributed Computing Systems*, pages 249–256, July 2002.