Mathematical and Computer Modelling 57 (2013) 2895–2903

Contents lists available at SciVerse ScienceDirect

Mathematical and Computer Modelling

journal homepage: www.elsevier.com/locate/mcm

Evaluation of contact synchronization algorithms for the Android platform

Vincent Seguí Pascual[a], Fatos Xhafa[b,*]

[a] Open University of Catalonia, Spain
[b] Technical University of Catalonia, Spain

ARTICLE INFO

Article history:
Received 22 July 2011
Received in revised form 12 October 2011
Accepted 22 December 2011

Keywords:
Data communications
Data Synchronization
Mobile computing
Intranet/extranet/VPNs

ABSTRACT

Synchronization protocols have been widely investigated in distributed systems aiming to achieve real-time and scalable properties. With the fast development of large-scale distributed systems, and due to their heterogenous nature involving wired, wireless, and mobile nodes, synchronization has again come into play. In this work, we have studied contact synchronization and handling, which is an important feature in corporate environments. Indeed, it has become very important to support collaboration of teams of mobile users by enabling anytime and anywhere access to shared contact data. We characterize the problem as a distributed systems problem, identify its desirable properties, and outline its main characteristics. A simple algorithm is proposed as an efficient solution to contact synchronization when some nodes of the system are assumed to be mobile phones under the Android operating system. The features required at both ends of the distributed system are explained in order to guarantee the correctness of the algorithm. We also analyze the implementation of the algorithm coupling the Android platform and the SugarCRM server, and provide an experimental evaluation of the performance of the proposed approach.

1. Introduction

Synchronization has been long investigated in different contexts of traditional distributed systems. One such context is that of data synchronization [1]. With the fast development of large-scale distributed systems, and due to their heterogenous nature involving wired, wireless, and mobile nodes, synchronization has again come into play. While known algorithms for data synchronization could be applied to distributed systems with mobile nodes, due to several features of such systems such as intermittent Internet connections, limited computational resources, battery restrictions, etc., it has become imperative to study them again and efficiently implement them for large-scale heterogenous distributed systems. In this work, we focus on the data synchronization problem on distributed systems with mobile nodes. The role of mobile systems has very quickly experienced a great growth in corporate IT infrastructure. Geographically distributed teams that need instant communication, coupled with the ubiquitous, albeit unreliable, presence of broadband Internet, have brought a new paradigm of business processes (and consequently of business software applications) in which the traditional client/server architecture needs to work with intermittent connectivity, and thus data synchronization is a basic feature in order to allow mobile systems to give service to the users in the absence of a direct connection to the central servers. Important use scenarios include mobile social networks to support the "*always-available*" features and that of mobile teams, namely, a group of people, geographically distributed and equipped with mobile devices, to collaborate for the accomplishment of

* Corresponding author.
E-mail addresses: vseguip@gmail.com (V.S. Pascual), fatos@lsi.upc.edu (F. Xhafa).

0895-7177/$ – see front matter © 2011 Elsevier Ltd. All rights reserved.
doi:10.1016/j.mcm.2011.12.039

a common project (e.g., a group of users of a Virtual Campus, a team of doctors in a disaster field, etc.) to support "*anytime*, *anywhere*" access to shared synchronized data.

The Android operating system has become a major player in the *smartphone* arena and as such it has an important ecosystem of software applications focused around it. In the corporate market, Android offers, out of the box, contact synchronization for the Exchange platform. Ironically, this leaves out a lot of companies that use Unix/Linux as their server platform. Google, however, provides a powerful framework that allows third parties to develop plugins for content synchronization and for managing multiple contact sources in the form of the Accounts, Synchronization, and Contacts API. At the same time, SugarCRM has become the market leader for open-source CRM solutions, making an integration between these two systems an interesting option for many applications and companies.

Synchronizers are difficult to specify and build due to the conflict resolution issues. While there are some examples of synchronization adapters, both in the wild as well as in the Android SDK, many of them are dedicated to single-user contact lists and to read-only adapters (e.g., the user is not able to modify the contact data through the Android interface), and they are, in this sense, of limited use.

Adding shared contacts and read/write access to the contact data in the *smartphone* changes the game considerably: it is no longer a matter of reading data from a server from time to time and updating our list accordingly, but, as we will see, one of building a distributed system similar to a document repository or a version control system. In this sense, a lot of work has been done in the form of several standards, the most prominent being SyncML [2]. This is a complex but very complete protocol specifically oriented to mobile systems, offering several synchronization modes, especially for synchronizing contacts and calendar information between mobile devices and servers. While it offers a complete solution to many synchronization needs, SyncML usually requires the presence of a third server in the system that integrates and manages all different data sources in a centralized fashion. This design allows for complex setups, but represents a burden to much simpler installations where the data source is present in one server and all we want to allow is for simple sharing. Furthermore, the setup of a third server can face opposition from system administrators and corporate policy. In this regard, the building of an *ad hoc* solution may be a better option. Some other works [3] assume a specific format such as XML tree-structures.

Vendors are also promoting the use of Cloud services for contact synchronization. A small comparison can be found at the Funambol (a synchronization vendor) website available by subscription [4]. While the use of this type of system is increasing, the fact that many companies are reticent to use cloud services to hold sensitive data makes this solution difficult to adopt for many companies.

In what can be considered the other extreme of the scale we find peer-to-peer (P2P) approaches [5] which try to solve the problem in P2P networks. Cohen [6] proposed a Java framework for P2P synchronization of object stores on mobile devices. While these approaches are interesting in their own right (for instance in building a P2P social network) they are not a good fit for the naturally centralized business IT infrastructure.

Other strategies currently in use involve the use of proprietary frameworks such as the Microsoft Sync Framework [7] or the use of distributed mobile databases (a small recollection of such tools can be found in [8]). Both of these solutions however require a system designed expressly to take advantage of the architecture and are thus difficult to apply to existing systems.

In this regard, we find that there are very few algorithms that allow for synchronization with existing server systems. Such algorithms are usually proprietary and still require some degree of change in the server side. In this regard, the algorithm proposed defines a clear and small set of requisites that a server system must provide in order to ensure correct synchronization. Such a set of requisites is minimized by providing conflict resolution in the client as opposed to the commonly used server implementation and by following design patterns from other distributed control systems.

The rest of the paper is organized as follows. We give a description of the contact synchronization problem in Section 2, where we also analyze classes of distributed systems and identify the desirable properties of our contact synchronization problem. The proposed approach for the contact synchronization problem and the analysis of its performance are presented in Section 3. We end the paper in Section 4 with some conclusions and directions for future work.

2. Description of the problem

As we have already advanced in the introduction, it is easy to see that two-way contact synchronization can be classified as a distributed system. We effectively have two or more nodes (a central server and one or more phone system) that access the same data (a contact list) to achieve some goal, namely, the effective dissemination of contact information as well as the propagation of changes in such data across the network.

2.1. Contact synchronization as a distributed system

Brewer's famous conjecture [9] later proved by Seth Gilbert [10] clearly states that distributed web services can not fulfill three properties, namely, consistency, availability, and partition tolerance, at the same time. This result, known commonly as the CAP theorem, puts a theoretical limit as to what can be achieved by distributed systems.

It is worth revisiting informally [11] what the consistency, availability, and partition tolerance properties mean in the context of distributed systems, since some contexts may give different semantics to the same words.

- Consistency is the property by which all nodes see the same data at the same time.
- Availability is the property by which any request made on the system will receive a response.
- Partition tolerance is the property by which the system continues to operate despite any number of lost messages.

It should be noted that Gilbert and Lynch's results hold in the presence of an asynchronous network model, as well as in the presence of a synchronous network model.

The importance of the CAP theorem resides in the insight and theoretical framework it has given to distributed system designers when it comes to building such systems, fostering a whole new (and not so new) set of designs that embrace such a theorem and aim to maximize these properties in some way or another (for instance the explosion of a new generation of database systems under the NoSQL banner).

While a good understanding of the theoretical grounds upon which the CAP theorem stands is of capital importance when designing complex systems, it is not our intention to overcomplicate the statement of the contact synchronization problem by the use of such theory. Suffice to say that it is a distributed system, and that our work as designers of a solution is to select the best tradeoffs possible.

2.2. *Classes of distributed systems*

A brief overview of current distributed systems can give us some insight as to what model, design, and tradeoffs can be found in current systems and can help us decide which ones best fit our requirements.

Let us start by taking a glance at Table 1. The table presents the CAP properties of several distributed systems. Note that we are not talking about specific systems but about the choices that many such systems make. For instance, some NoSQL databases may actually not be tolerant to partition at all, some VCS may offer some limited support when working offline, and so on.

Also note that some systems offer a limited sense of consistency, usually called eventual consistency, even in the presence of the other pair of properties of the CAP triad. While ordinary consistency refers to the property that all nodes see the same data (or in some definitions the same set of operations) all the time, eventual consistency refers to the property that all nodes will see the same data or set of operations at some point in time, thus the name eventual. This sort of design has been coined with the acronym BASE, *Basically Available, Soft estate, Eventually consistent* by Pritchett [12] of eBay, and constitutes an important practical result derived from the theoretical framework provided by Brewer's theorem. The work of Pritchett, however, is heavily geared to the design of huge web backbones and complex systems.

Next in the list we can see a regular (e.g., centralized)Version Control System (VCS). Such systems usually take the form of centralized client/server systems and are basically not tolerant to network partitions. They are eventually consistent, and this is an important point, not because of any inherent limitation (after all, if they are not partition tolerant, they could be made to be consistent and available) but because the users of such systems usually only want to share changes once they consider them complete (at some point in time).

Adding partition tolerance to traditional VCSs will render us a DVCS. In this design, each user hosts a complete repository, and changes are propagated in many ways, sometimes by using a central repository or sometimes by sending to peers; the design allows for many different workflows. A particular property of these systems is that, expanding upon the lack of consistency of a centralized VCS, a DVCS does not even aim at being eventually consistent. Effectively, one of the advantages a DVCS claims is the ability of all users to have their own branches, patches, and history of the repository. This obviously means that, generally speaking, DVCS users have a different view of the data and thus the data are never consistent across all the nodes. Obviously this does not mean that *all* data are *never* consistent (developing anything useful in such an environment would be unlikely), but rather that only *some* data are *eventually* consistent. Normally such subsets of data are public branches where developers integrate and propagate their changes. Thus, if eventual consistency relaxed the time constraint on the consistency property, DVCSs further relax the data constraint in such a way that only part of the data will be eventually consistent or, in brief, partially consistent.

Other examples are NFS and P2P systems. The former are usually designed under the assumption of a reliable network and the latter assume the operation of all users as equal, avoiding the use of a central server. Both of these conditions however are undesirable for our system since mobile systems have limited network connectivity and corporate environments are usually designed around a set of servers.

2.3. *Desirable properties of contact synchronization*

We have seen several examples of distributed systems and the choices and tradeoffs made by their designers. These lessons should prove useful to the design of a contact synchronization system. Clearly, our requirements include partition tolerance and availability: users expect to access their contact list even without a network connection and expect it to be available even when other nodes may be down. Our main design decision, then, would be what guarantees of consistency we want our system to fulfill.

At first glance, one may think that eventual consistency is desirable, but, in such a model, the deletion of a contact in a mobile phone will lead to the deletion of the contact first in the server, and this will slowly propagate to all phones and clients. This obviously can lead to severe security incidents if the mobile phone is lost or used by an attacker, who could

Table 1
Distributed systems and their properties.

System	Partition tolerance	Availability	Consistency
NoSQL	Yes	Yes	Eventually
VCS	No	Yes	Eventually
DVCS	Yes	Yes	Partial
MPI cluster	No	Yes	Yes
NFS	Limited	Yes	Mostly
P2P	Yes	Yes	No

delete all contacts from the system. Partial consistency can solve this issue by ignoring deleted contacts at both sides of the system. There are other use cases in which partial consistency may be of value: for instance, users may want to attach different metadata to the same contacts (in which group a contact should be displayed in the phone).

We can conclude, then, that the desirable properties for our system would be to be partition tolerant, available, and partially consistent.

3. Solving contact synchronization

We have seen that the DVCS design apparently fits better in our design choices; however, it is also true that DVCSs offer features like the ability to synchronize between two peers without the use of a server. While this feature is certainly useful, keeping track of what changes have been propagated and to whom can be a complex endeavor and can pose a burden to the user in the form of data conflicts, change tracking, and other issues which make sense in the context of a DVCS but not for the problem at hand. What we really want is more akin to a VCS system with the twist that, like in a DVCS, not all data should be synchronized. This is obviously much easier to implement than a full-blown DVCS system.

Another important source for inspiration is obviously the SyncML protocol. The protocol defines a set of seven distinct algorithms for data synchronization. Of them, only two offer two-way synchronization, the first being "two-way sync" and the second being "slow-sync".

SyncML's version of two-way sync starts by sending the client's changes to the server for them to be processed. The server processes such changes and then sends a list of the accepted or final changes along with a set of its own changes back to the client. This algorithm obviously works well but needs an intelligent server able to do automatic conflict resolution and other data processing. However, our system is designed under the assumption that the server is not specifically designed to perform the contact synchronization task. Under this constraint, conflict resolution must be performed on the client, as our server may not be capable of performing such function.

The "slow-sync" algorithm is an expensive algorithm in which all elements in both databases are compared against each other. We will not provide a detailed description in this paper, but it suffers the same problems that the ordinary "two-way" algorithm has.

3.1. A simple algorithm

As we have seen previously, the VCS design is perhaps the best suited to our problem. The algorithm given as Algorithm 1 should not come as a surprise. Indeed, we are mimicking the usual workflow and algorithms that users of CVS, Subversion, and other VCSs use daily.

The algorithm uses a similar notion of "Anchor" that can be found in SyncML, namely, an object that represents a successful synchronization operation in time. How time is represented (as a date, as a version number, etc.) is left to the server system to decide.[1] Unlike SyncML, however, the last successful anchor is saved in the client instead of in a table in the server.

The algorithm works as follows. The client requests all modified contacts since the current anchor it has stored. If no anchor is found, all contacts are retrieved from the server. In the next step it gets all locally modified contacts since the last successful synchronization operation. In Android, this information is stored for each contact as a flag that marks it as "dirty" any time the user modifies such a contact. Once we have the set of modified contacts in the server and modified contacts in the client, we obtain the intersection of both sets. Elements in this set are candidates for conflict; elements out of this set are non-conflicting. Four sets of contacts are then calculated.

- C_{cs}: The set of contacts from the server that are really conflicting (actually have different data) with locally modified contacts.
- C_{cl}: The set of modified local contacts that are really conflicting (actually have different data) with remotely modified contacts.

1 In SugarCRM, anchors are represented by dates.

Algorithm 1 *synContacts*(*lastSync* : *Anchor*) : *Anchor*

```
$C_{server} := server.getNewerContacts(lastSync)$
$C_{local} := local.getLocalModifiedContacts()$
$C_c := C_{server} \cap C_{local}$
$C_s := C_{server} - C_c$
$C_l := C_{local} - C_c$
$C_{cs} := \emptyset$
$C_{cl} := \emptyset$
for $c \in C_c$ do
  $c_l := pop(C_{local}, c.id)$
  $c_s := pop(C_{server}, c.id)$
  if $c_s \neq c_l$ then
    $C_{cs} := C_{cs} \cup \{c_s\}$
    $C_{cl} := C_{cl} \cup \{c_l\}$
  else
    $C_s := C_s \cup \{c_s\}$
    $C_l := C_s \cup \{c_l\}$
  end if
end for
$C_r := resolveConflicts(C_{cs}, C_{cl})$
$C_s := C_s \cup C_r$
$C_l := C_l \cup C_r$
$server.sendContacts(C_l)$
$local.saveContacts(C_s)$
$local.markCleanContacts(C_l)$
$currentSync := max(\{d | d = c.anchor\ \forall c \in C_s\} \cup \{lastSync\})$
return $currentSync$
```

Table 2
Testing configuration.

Processor	Intel(R) Core(TM)2 Duo CPU T7500 2.20 GHz
Ram	4 GB Ram
Operating system	Ubuntu 10.10 × 64
Server platform	SugarCRM 6.1 running under VirtualBox
Virtual OS	Ubuntu 10.10 × 64
Client platform	Stock Android emulator running 2.1 and SDK 10

- C_s: The set of modified server contacts that are not conflicting with the set of locally modified contacts.
- C_l: The set of modified local contacts that are not conflicting with the set of contacts obtained from the server.

From these four sets, C_{cs} and C_{cl} are presented to the user for conflict resolution (see Fig. 1), and a resolution set C_r is created and then joined with C_s and C_l. We then send the new C_l to the server (after it is successfully sent to the server, C_l is marked as clean). Finally, C_s is stored locally (and marked as clean in the same process).

3.2. Performance analysis

As we can see, the algorithm is efficient in the sense that, generally speaking, it only transmits the needed contacts between both systems, and, more importantly, the number of round trips between the server and the client is fairly low (between 2 or 3 HTTP round trips depending on whether the session is active or has to be started anew). Measurement of the performance, however, is difficult, due to the heterogeneous nature of the platform and the network connections. Wall clock time can vary wildly when run with the emulator or when run with an actual phone (the same can be said when run between different phones). As a rule of thumb, a full synchronization of 200 contacts can take up to 270 s of wall time (0.77 contact/s) when performed in the emulator. This includes network and storage time when the emulator is running in the same physical machine as the SugarCRM that is running in a virtual machine. In contrast, when executing the same operation in an *HTC Desire HD* phone, the wall time is around 45 s, yielding 4.6 contacts/s or a sixfold speedup.

Given the different performance characteristics, for testing purposes, we will use the Android emulator, as this is the easiest setup to reproduce. Our testing configuration can be seen in Table 2.

We will test how the algorithm scales with the number of contacts (we will execute a full synchronization with contact lists ranging from 200 to 4000 contacts). Also we will try to identify potential bottlenecks which will point us to future optimizations.

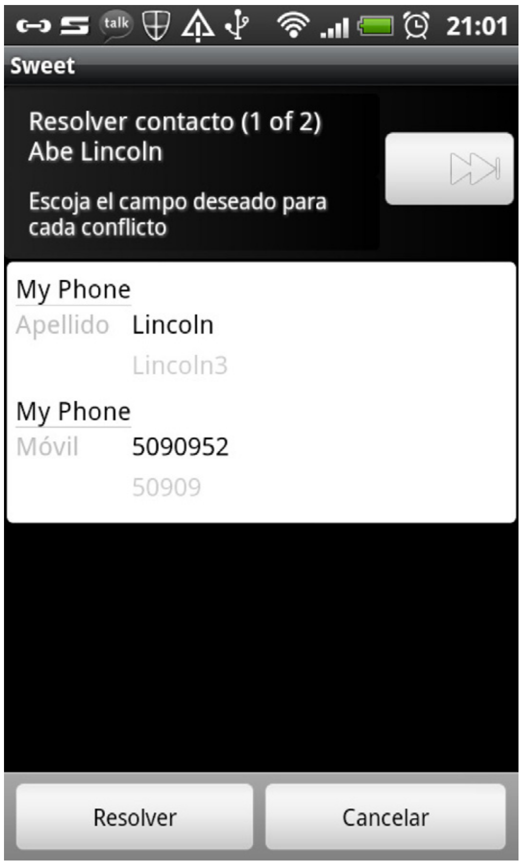

Fig. 1. User interface for conflict resolution.

Table 3
Timing results.

List size	Pristine sync operation		Full sync operation	
	Total (s)	Average (c/s)	Total (s)	Average (c/s)
200	52	3.84	125	1.6
400	98	4.08	484	0.83
600	150	3.99	712	0.84
800	230	3.47	931	0.85
1000	415	2.40	1093	0.91
2000	752	2.66	2227	0.90
4000	2002	2	5787	0.69

The data used for performance are those included in a stock SugarCRM installation which yields around 200 contacts. These data have been duplicated as needed to achieve the whole set of contacts. In Table 3 we can see the wall clock time results of several contact synchronization operations. Two different operations are performed for each list size. The first is a pristine full synchronization operation. The second is a forced full synchronization operation once the data have already been loaded in the Android phone database. For each of these two operations two results are measured and reported: the total wall time elapsed and the average amount of time per contact.

Two charts with the results are also given for both metrics, total and average time (rounded to second accuracy), as can be seen in Figs. 2 and 3.

A seemingly surprising fact can be immediately observed from the gathered data: pristine synchronization operations are faster than forced full synchronization. The reason why this is so is due to the way our synchronization algorithm stores data in the Android contact database. Generally speaking, every field in a contact corresponds to one row. In fact this is a simplification, since two fields in the SugarCRM contact may correspond to only one row with two columns in the Android database. The mapping can vary according to the type of field. In the case of a pristine synchronization, the algorithm is sure that no contact exists in the database so it can issue several insert commands in batches, which is a fast operation. When doing a full forced synchronization operation, the algorithm first has to check if the contact exists in the database. If it does, it has to issue an update command; if it does not, it will have to issue an insert command. Additionally, for each existing contact,

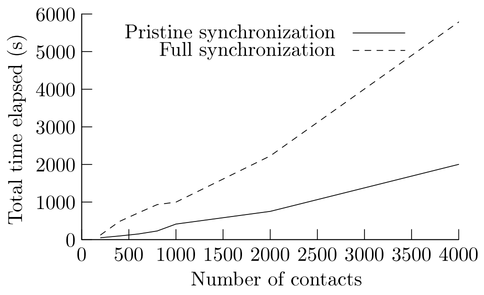

Fig. 2. Wall time performance.

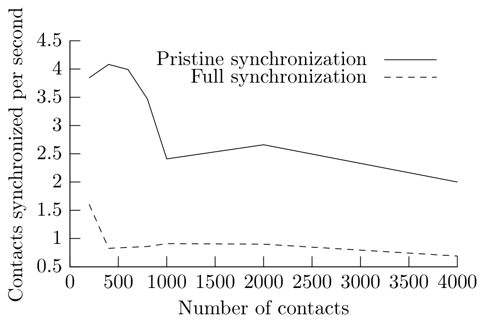

Fig. 3. Average performance.

we have to check whether the fields exist and issue update or insert commands accordingly. It should be noted that while SQLite databases do support "upsert" operations, the standard Content Provider API does not, making these operations more cumbersome than they really need to be. This means that full forced synchronizations actually have to perform many more commands than ordinary operations, mixing read operations with write operations and thus degrading the performance. There are several fixes to this. For instance, inserting empty rows for each empty field when first creating a contact. Once done, we can always issuing an update command avoiding field lookups when updating a contact. This comes at a price in persistent space, so careful analysis of the tradeoff has to be made since Android devices can be low on space. Another solution would be to get all non-modified fields in a batch and compare them to the server list, removing from the update list those that are equal. Again, careful analysis has to be made as this tradeoff uses more RAM to hold both lists, which can be a limited resource in mobile systems. When pondering both of the above strategies, we also have to take into consideration that a forced full synchronization operation should be an exceptional operation, and that a regular synchronization will only involve the list of modified contacts instead of the whole list, making the problem much smaller. Future work and analysis in this regard is thus warranted in order to make the correct decisions.

As for the scaling characteristics of the algorithm, we can see that it behaves linearly with the size of the contact list (as expected) until we hit the 1000 contacts list size,[2] except for the pristine synchronization of the 200-sized contact list. This difference can probably be attributed to the Android GC kicking in, but further investigation is needed as to why there is such a degradation between the 200 and 400 contact list sizes. At this point, there are two issues to address. First, the stock installation of SugarCRM and PHP has to be modified to allow for more memory in order to avoid internal server errors. On the Android side, the VM heap has to be increased to avoid out of memory errors when parsing the response. The behavior continues to be more or less linear until the 4000 contact list size. In order for the algorithm to perform correctly, the emulator device has to be given more RAM. A degradation for the performance of both types of synchronization can be observed although it is difficult to ascertain if this is due to measurement error, GC, or even server-side problems. While the algorithm does not scale badly, it can be seen that it suffers with large contacts list requiring adjustments in the heap size of the Android end and in the server. Two approaches can be used to tackle this problem. A first one would be using a streaming API that parses the input as it is read instead of buffering it all in memory. This, however, would require a different program architecture and probably third-party libraries to allow for JSON streaming parsing. A second approach would download contacts in batches of a few hundred instances and parse them before downloading the rest. This later approach has the drawback of requiring more round trips, but it does seem as workable if special care is taken to download the contacts in a proper order that guarantees no lost updates.

[2] Obviously the list size at which this happens can vary depending on the data stored for each contact, the stock installation, the operating system, etc.

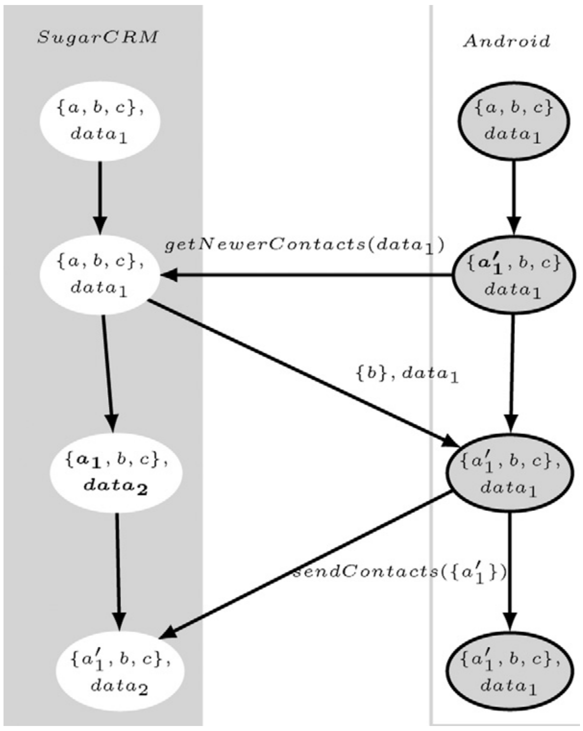

Fig. 4. Lost update problem.

3.3. *Correctness of the algorithm*

While not a formal proof, it is interesting to show what features the server and the client must support in order for the algorithm to behave correctly. We can see that, without much analysis, there are flaws in the algorithm that can cause loss of data. For instance, consider the sequence depicted in Fig. 4. In this scenario, a user modifies contact a, creating contact a_1 in the server after a client has requested the list of modified contacts (along with the anchor labeled as $data_1$). This contact has also been modified at the client side, creating a'_1, but contact a_1 has not been transmitted to the client, and therefore the client is unable to detect the conflict. The client then sends this modified contact to the server, which blindly overwrites the a_1 contact with the a'_1 contact, causing the loss of data.

Clearly, we need some support from the server in order to avoid this problem. Note that no algorithm can possibly work correctly as long as the server does not offer any sort of support for detecting conflicts or locking. This requirement is not as strong as it may seem: servers must support some kind of mechanism to avoid this problem since it can happen when clients access the server via the Web or a desktop interface (also note that this feature is a much lighter requirement than conflict resolution, which was one of the reasons we discarded SyncML algorithms where the server had to offer this capability). In the case of SugarCRM, optimistic locking is used, whereas the server will check for conflicts when committing the changes. This kind of setup is a perfect match for our use case (low data contention): if SugarCRM manages to warn us of the error when we send it the list of changes, we can cancel the operation and retry later (very much like a failed transaction is retried at a later time). This is possible since at this point the algorithm has not realized any side effects (e.g., it has not changed either the local or remote database). Unfortunately, though, SugarCRM does not seem to go through the optimistic locking when being invoked by REST or SOAP calls. This has been reported as bug 44316 [13] in the SugarCRM bug tracker.

Similar issues can affect the local contact database; however, these can easily be solved by locking the data. This is possible because, under the Android platform, contact editing must be made available by the synchronization developer; thus we can choose to disable it while synchronizing. With this setup we can avoid the lost update problem in the client.

Once we have committed the data to the server, how can we guarantee that the operation will complete? The answer is that it does not really matter. Supposing we fail, the next time we try to synchronize, the anchor will still be the old one, and we will pull all the new changes (including those *we sent* to the server) plus all the old changes. Since our algorithm is smart enough to distinguish real conflicts by comparing the actual contacts, we will not get any conflicts on those contacts. It should be noted that special care has to be taken for locally created contacts that have been sent and created remotely but, due to failure or other reasons, not linked with the locally created contacts. In this case, before sending newly created contacts, we check if they already exist in the server list and link them accordingly if so. However, this is transparent to the user as it is done automatically by the *resolveConflicts* operation. From the user's point of view, everything will work as usual, and no special actions need to be taken even if some bandwidth will be wasted by retransmitting needless data.

Finally, there is a last issue of what happens to deleted contacts. As we can see from the algorithm, no special action seems to be taken. This is because of the way deleted contacts work in both systems (SugarCRM and Android). When a contact is deleted it is not really deleted from the database in either platform, but rather flagged as a deleted contact. By choosing not to synchronize the said flag we accomplish the requirement that deleted contacts do not propagate across the system. This can easily be extended to any other data or metadata we may wish to keep out of the consistent data set, so, for example, group membership or company affiliation could be left out of the synchronization process.

4. Conclusion and future work

Mobile computing is a strong reality in today's IT infrastructure, and the need to share data and make it available everywhere, at any time, across multiple devices operating under unreliable conditions is a challenge that can be tackled with different approaches.

We have presented an example of a distributed system involving mobile systems together with the rationale that has led to this design by observing other systems and applying the principles embodied in the CAP theorem (consistency, availability, and partition tolerance) and related work. We have also classified this kind of design in a category of its own that has partial consistency as a desirable feature. We have shown that the system can work correctly and reliably as long as some basic support is offered by the server system, even in the event of network or device failure. This comes at the cost of some redundancy when transferring data after failure.

Future work will be oriented towards two different directions. The first will be measuring actual performance and bottlenecks in the current system across several devices. This will allow us to optimize the real bottlenecks regardless of the device. The second will be concerning the synchronization of calendar data, which, unlike contact data, can reap the benefits of a fully blown distributed design by enabling users to send events across different networks (SMS, email, etc.) without the need to have connectivity with the server which may be difficult to achieve (for example with roaming users who may only have SMS as a viable transport).

References

[1] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, Alan Schmitt, Exploiting schemas in data synchronization, Journal of Computer and System Sciences 73 (2007) 669–689.

[2] D.-W.K. Byung-Yun Lee, Tae-Wan Kim, H. Choi, Data synchronization protocol in mobile computing environment using syncml, in: 5th IEEE International Conference on High Speed Networks and Multimedia Communications, 2002, pp. 133–137.

[3] T. Lindholm, XML three-way merge as a reconciliation engine for mobile data, ACM Symposium on Document Engineering (2004) 1–10.

[4] Funambol Inc., Evaluating Mobile Cloud Sync Solutions: From Apple MobileMe to Vodafone Zyb.

[5] J. Vogel, W. Geyer, Li-Te Cheng, M. Muller, Consistency control for synchronous and asynchronous collaboration based on shared objects and activities, Computer Supported Cooperative Work 13 (2004) 573–602.

[6] N.H. Cohen, A java framework for mobile data synchronization, in: Cooperative Information Systems: 7th International Conference, CoopIS 2000, in: Lecture Notes in Computer Science, vol. 1901, Springer, 2000, pp. 287–298.

[7] Microsoft Sync Framework. URL http://msdn.microsoft.com/en-us/sync/bb821992.

[8] Wikipedia, Mobile Database. Online, URL http://en.wikipedia.org/wiki/Mobile_database, 2011.

[9] E.A. Brewer, Towards robust distributed systems, invited talk, in: Symposium on Principles of Distributed Computing, Portland, Oregon, 2000.

[10] N.L. Seth Gilbert, Brewers conjecture and the feasibility of consistent, available, partition-tolerant web services, ACM SIGACT News 33 (2) (2002) 51–59.

[11] Wikipedia, Cap theorem. Online, URL http://en.wikipedia.org/wiki/CAP_theorem, 2011.

[12] D. Pritchett, Base: an acid alternative, ACM QUEUE 55 (2008) 48–55. http://queue.acm.org/detail.cfm?id=1394128.

[13] Sugarcrm bug 44316. Online, URL http://www.sugarcrm.com/crm/support/bugs.html#issue_44316.