Technical Report TRCS00-07
Planned Disconnections for Mobile Databases *

JoAnne Holliday   Divyakant Agrawal   Amr El Abbadi
Department of Computer Science
University of California at Santa Barbara
Santa Barbara, CA 93106
{joanne46,agrawal,amr}@cs.ucsb.edu

Abstract

As mobility permeates into todays computing and communication arena, we envision application infrastructures that will increasingly rely on mobile technologies. Traditional database applications and information service applications will need to integrate mobile entities: people and computers. In this paper, we develop a distributed database framework for mobile environments. A key requirement in such an environment is to support frequent connection and disconnection of database sites. We present algorithms that implement this framework in an asynchronous system.

1 Introduction

As mobility permeates into todays computing and communication arena, we envision application infrastructures that will increasingly rely on mobile technologies. Current mobility applications tend to have a large central server and use mobile platforms only as caching devices. We want to elevate the role of mobile computers to first class entities in the sense that they allow the mobile user work/update capabilities independent of a central server. In such an environment, several mobile computers may collectively form the entire distributed system of interest. These mobile computers may communicate together in an ad hoc manner by communicating through networks that are formed on demand. Such communication may occur through wired or wireless networks. At any given time, a subset of the computer collection may connect and would require reliable and dependable access to relevant data of interest.

In this paper, we consider a distributed database that can be made up entirely of mobile components. These component sites are peers and may be replicated both for fault-tolerance, dependability, and to compensate for sites which are currently disconnected. Thus we have a distributed replicated (partial or total) database where several sites must participate in the synchronization of transactions. The capabilities of the distributed replicated database are extended by allowing a site or team member to plan to disconnect so that the remaining sites are minimally disrupted. A "check-out" mode is described that allows for independent update of a portion of the database by a disconnected member. These updates are automatically synchronized and integrated into the database upon reconnection. A "relaxed check-out" mode permits increased concurrency at the expense of serializability. An implementation in an asynchronous system is discussed. Our approach emphasizes the mobile aspect of all system components, and hence the lack of a centralized server.

Walborn and Chrysanthis [12] describe the use of mobile computers in the trucking industry. Each truck has a computer with a satellite or radio link and interacts with the corporate database. Other applications involving remote or disaster areas and military applications have mobile computers forming Ad-Hoc networks without communications with stationary computers. Faiz and Zaslavsky [3] discuss the impact of wireless technologies and mobile hosts on a variety of replication strategies. Distributed replicated file systems such as Ficus and Coda [8] have extensive experience with disconnected operations. In this paper, we accommodate voluntary disconnection, by using the notion of planned disconnection, which has been shown to be of particular use in asynchronous systems [6]. We present an algorithm to allow disconnected update using a check-out idea adapted from version control software. In Section 2, we present three alternative disconnection modes. In Section 3, we present our model of a distributed database system. In Section 4, we explain in detail how the three modes will operate. In Section 5, we show how to implement the modes in an asynchronous system. We conclude with Section 6.

* This research was partially supported by the NSF under grant numbers EIA98-18320, CCR97-12108, IIS98-17432 and IIS99-70700.

2 Motivation

In the system we are considering, the components of the distributed database are laptop computers belonging to members of a small team. The team members maintain a database which may or may not be replicated at each of their laptop computers. Normal operations ensure that all team members are completely synchronized with each other's progress. Team members gather together in various locations and form ad hoc networks to perform their work. These ad hoc networks may be wired or wireless. Several varieties of wireless LANs are already available. Examples include NCR's wavelan (Lucent Technologies), Motorola's ALTAIR, Proxim's Range LAN, and Telesystem's ARLAN. The laptop computers of the team members carry the only active copies of the database. When team members gather in a remote location, they need to have update capabilities on their database without having to connect to a fixed location computer. The database system must deal with both planned and unplanned disconnections.

We consider three scenarios covering ways in which the team members might want to interact with their database: basic sign-off, check-out, and relaxed check-out. In the first scenario, team members occasionally disconnect for long periods of time, for instance when one or more of them are traveling. When this happens, the remaining team members need to be informed so that the database can continue to process updates knowing that the missing processor has not failed but rather has voluntarily disconnected. The database copy of the traveling member can continue to process read-only transactions. We call this mode *Basic Sign-off*. The idea of planned disconnection is not new [6] and it is reasonable for computers in a distributed database to go through a sign-off procedure before disconnecting to tell the remaining members not to expect acknowledgments or votes on the disposition of shared data.

Our first scenario involves Bob, who is the CEO of a small company, in fact he has only 11 employees. It was decided to spend the "office equipment" budget getting laptop computers for the three senior executives and a desktop computer for the executive secretary. The three laptops of the senior executives contain replicas of the corporate database, which is generally updated during their frequent meetings when they connect their computers with an ad-hoc network. The database is backed-up occasionally onto the secretary's computer. When Bob needs to travel to meet with customers or give a presentation, he wants the other senior executives to be able to continue their work with the corporate database in his absence. He also wants to take his laptop on his trip so that he can peruse the database while he is waiting at the airport or in his hotel room. For the basic sign-off mode, the database system provides a sign-off capability for Bob when he or another one of the senior executives wishes to travel, otherwise, the distributed database system would not be able to synchronize all of the database components and wouldl not allow updates. A manual override can handle the case where a member has forgotten to sign-out. When Bob returns and reconnects, he wants his copy of the database to automatically be brought up to date. For this, the database system provides a sign-on procedure.

In the next scenario, the traveling team member wishes to "check-out" a portion of the database so that updates can be made to that portion while disconnected from the rest of the members. Thus, this mode is called *Check-out* mode. This mode could be needed when the team member is traveling to a location where he will be able to gather information relevant to that portion of the database or perhaps that portion of the database is concerned with the work or output of that particular team member. When the traveling team member has checked-out a portion of the database, the other team members are completely unable to access that data. It is as if the traveling team member had a write lock on the data that was checked-out. The traveling member can access in read-only mode other portions of the database that he did not check-out. When the traveling team member returns and reconnects his laptop to the distributed database, the database changes he has made are automatically synchronized and integrated into the database. This is in contrast to a system without check-out mode that requires manual integration of the changes into the database.

An example of the check-out mode would involve Alice, who is part of a small sales team that currently covers the pacific northwest states. She meets with her team almost every day at which time they bring their laptops together to form a network and share their information. At that time, their common database is updated with production and product availability information from the factory. Before the computers are disconnected, each team member is given a sales assignment and a portion of the database is checked-out to them to update as they make sales and delivery contracts. Alice can update the portion that is checked-out to her as well as query the portions of the database that were not checked-out to anyone. She cannot access the portions of the database that were checked out to other team members, however, that is not necessary for her to do her work. At the next meeting, the laptops are connected and the updates to the database are smoothly integrated without the need of manual intervention.

The third scenario, *Relaxed Check-out* mode, is like check-out except those portions of the database that have been checked-out to other team members can be queried, that is, accessed in read-only mode. Unlike the above two scenarios where all transactions are serializable, the resulting transactions are not serializable. However, there are many situation in which a relaxation of serializability requirements is acceptable.

Jane is a field researcher in a remote part of the world. She and her fellow researchers take their laptops to remote observation platforms and weather stations. They record animal migration patterns and download weather information from sensor stations. While she is recording animal migrations she frequently needs to check recordings of an earlier day or a different location to help her in interpreting her measurements for her field notes.

The database on the computers of Jane and her fellow researchers allows relaxed check-out mode. She checks out the portion of the database that pertains to the location she is going to or the animal species she is studying. She and her fellow researchers can also query the portions that are checked-out to other researchers, although the query results reflect old values from the last time the researchers gathered together and formed a network. She knows she may be viewing stale data, but it is still useful to her to do so. When she checks-in her work at the end of the day, it is automatically integrated into the database.

3 Database Model

We consider a distributed system consisting of n sites labeled $S_1, S_2, ..., S_n$. The database may be partially or fully replicated, that is, data items are stored redundantly at multiple sites. Information about the location of all copies of a data item may be stored at each site or kept in directories at several of the sites. Users interact with the database by invoking transactions at any one of the database sites. A transaction is a sequence of read and write operations on the data items that is executed atomically. The criterion for correctness in databases is the *serializable* execution of transactions [2]. Serializable executions are guaranteed by using a *concurrency control* mechanism such as two-phase locking, timestamp ordering, or optimistic concurrency control [4]. Since two-phase locking is widely used, we assume that each site in our distributed system enforces two-phase locking locally. In a replicated database, multiple physical copies must appear to the user as a single logical copy. Therefore, the criterion for correctness in replicated databases is *one-copy serializability* [2]. We assume a mechanism to enforce one-copy serializability is used. It could be a synchronous control protocol using Read-One-Write-All (ROWA), for example, or an asynchronous site management scheme such as the one discussed in section 5. In the interest of generality, we do not require any particular communication style or distributed database management scheme as long as it has some form of global synchronization.

In this paper, we deal with a distributed databases in which:

- The number of team members is fixed, however, some of those team members may be disconnected. In fact, members of the distributed system frequently disconnect. Members will need to gather and form an ad hoc network in different geographical locations.
- Database consistency is important. Serializability is enforced locally with two-phase locking. Some method of global synchronization must also be used, although to preserve generality, we will not specify which method.
- The distributed system must be able to process database updates even though some of the team members are not available. It is not desirable to designate one of the members as the primary site which must always be present when a network is formed, therefore all members should be equal participants.

4 Planned Disconnection Types and Implementation

Unplanned disconnection is a disconnection without informing the distributed system of the intention to disconnect and reconnect in an orderly manner. The result is that the disconnected site is detected as a failure. Planned disconnection involves informing the distributed system of the intention to disconnect and may include the appointing of a proxy. The purpose of a planned disconnection procedure is to enable the remaining connected sites to continue processing with minimal disruption. In this section, we explore the possible kinds of planned disconnection and provide appropriate terminology.

4.1 Basic Sign-off

In simple planned disconnection or *basic sign-off* [5], the database of the disconnected site becomes read-only. The connected sites continue to read and update the data item or objects. A ROWA system adjusts its list of replicas so that update transactions can complete in spite of a missing member. The planned disconnection will be accomplished with the help of a *proxy*. When a site disconnects, it appoints another site to vote on its behalf to ensure that replicas can be updated and in any other actions of the distributed system which require consensus. The power to vote on behalf of another member is called a proxy and the site with that power is also referred to as a proxy.

If site S_i wants to disconnect from a distributed database, it carries out a disconnect dialog so that the system is aware that it has not failed, but will merely disconnect for a period of time. S_i contacts another member S_j to be S_i's proxy. During the disconnection, S_i can only read its local copy of the data. When S_j sees a message for S_i, it answers on behalf of S_i while S_i is disconnected. S_j also keeps track of the updates to the database that S_i has missed because of the disconnection. (We assume that all updates are sent to all copies, and so, S_j can respond on behalf on S_i.)

Assume that site S_i wishes to disconnect. The sign-off procedure is as follows:

1. Site S_i selects a proxy. The proxy can be any peer which is currently connected or a special site designated to hold proxies.

2. For those data items at site S_i that are not replicated at another currently connected site, give the proxy a copy of the data items and the right to process update transactions on them. The data items at S_i will become read only like the rest of the database. If the information concerning the location and connection status of replicas is not readily available, S_i must send out messages to determine such information.

3. For those data items at site S_i that are replicated at other sites, give the proxy the right to vote for S_i in matters concerning updates to the data items.

4. Ensure that the proxy is aware of S_i's current state, i.e., which updates have been processed and which have not and then disconnect

S_i must also go through a sign-on procedure when reconnecting:

1. Reconnect and contact the proxy before processing messages from other sites. If S_i's proxy has itself disconnected, broadcast messages to determine who the new proxy is.

2. With the help of the proxy, determine what updates need to be made to the database to bring S_i up-to-date and incorporates those updates.

3. Retrieve voting rights from the proxy, release it from duty and then resume normal operations.

In the case where all sites are disconnecting and S_i is the last to sign-off, there is no other site for S_i to give its proxy to. In this case, S_i simply disconnects without assigning a proxy as no one in the distributed database system will be performing updates or anything else, for that matter.

The Basic Sign-off protocol produces executions that are one-copy serializable. Briefly, it can be argued that, since all of the transactions of the disconnected site are read-only, the values of the data that are read are those of a snapshot taken at the time of disconnection. All of the read-only transactions of the disconnected site can be serialized at the time of disconnection.

4.2 Check-out mode

If the site S_i wants to disconnect and be able to update a particular data object, it declares its intention to do so before disconnection and "checks-out" or "takes" the object for writing. This might be accomplished by obtaining a lock on the item before disconnection. In order to maintain serializability in *check-out* mode, the remaining connected sites are prevented from accessing the object which S_i has checked-out (as if S_i had a write lock). An object can only be checked-out to one site at a time.

Since many database systems use two phase locking, it makes sense to implement *check-out* mode using the existing locking mechanisms. The site that wishes to disconnect, say, S_i, acquires a write lock on the item or object it wants to update while disconnected. This write lock is like an ordinary write lock except that the "transaction" which holds it (if there is one) should not be aborted due to deadlock with ordinary transactions. The mechanism for obtaining the lock might be via a transaction or through some other means. In order to distinguish these "transactions" from ordinary user transactions, we will call them *pseudo-transactions*.

Assume that site S_i wishes to disconnect and "check-out" a set of items X. The disconnect procedure goes as follows:

1. Site S_i selects a proxy as in basic sign-off mode and follows all the same steps to handle voting rights for replicated and non-replicated items.

2. At the same time, S_i initiates a pseudo-transaction to obtain write locks on the items in X.

3. If the pseudo-transaction is successful, S_i disconnects with update privileges on all the items in X. If the pseudo-transaction is not successful, S_i tries again or disconnects with update rights on a subset of the items.

When S_i reconnects, the reconnect procedure is the same as for basic sign-off except that S_i must complete the effects of the pseudo-transactions by transmitting the new value for all items in X and releasing the corresponding locks.

Figure 1 shows the activity at three sites, S_i, S_j and S_k. Time proceeds from left to right and an asterisk (*) indicates the disconnection and reconnection points in time. X_i indicates the version of data item X written by transaction i. In Figure 1, S_i first acquires a write lock on X with a pseudo-transaction (pt) and disconnects. t_1 and t_2 are examples of transactions that may be executed at the disconnected site S_i. Site S_j later disconnects with a write lock on Y and executes transactions that read Y and Z and update Y, e.g., t_4. Site S_k remains connected and executes transaction t_3. Notice that S_i can execute transactions during its

S_i	*pt: wl-X & disconnect				*reconnect & $w(X_2)$ c
	t_1 $r(Y_0)\, r(Z_0)\, w(X_1)$ c				
			t_2 $r(Y_0)\, r(X_1)\, w(X_2)$ c		
S_j		*pt: wl-Y & disconnect		*reconnect & $w(Y_4)$ c	
		t_4 $r(Y_0)\, r(Z_3) w(Y_4)$ c			
S_k					
	t_3 $r(Z_0)\, w(Z_3)$ c				

Figure 1. Check-out Mode with Write Locks

disconnection that read all other database items without getting read locks before disconnection on those items. This is because all of S_i's transactions will be reading versions of those data items that existed at disconnect time. In order to preserve correctness, it must be possible to serialize all of the transactions executed by S_i during disconnection at the point in time of disconnection. This can be done if:

1. Only those items write locked by pseudo-transactions at disconnect time can be modified by S_i during disconnect.

2. Those items write locked by pseudo-transactions at disconnect time can neither be read nor written by other sites (consequence of holding write lock) and the pseudo-transaction can not be aborted in order to release the lock.

3. Items not write locked by pseudo-transactions at disconnect time are treated as read-only by S_i during disconnect (unless they were currently locked by other transactions at the time of disconnection).

This will guarantee serializability because each transaction whether at a connected or disconnected site respects two phase locking. The disconnected site is required to respect locks held by on-going transactions from other sites at the time of disconnect. Thus in Figure 1, t_4 executes under the conditions that X is write locked by a pseudo-transaction and other items may be locked by transactions that were on-going when site S_j disconnected. The equivalent and correct serial order for these transactions is t_1, t_2, t_3, t_4.

The Check-out mode produces executions that are one-copy serializable. Briefly, write locks obtained at the time of disconnect by pseudo-transactions are held during disconnect and released only after the new values are revealed at reconnection time. These locked items are the only data items modified by the transactions at the disconnected site and these transactions are serialized with respect to each other because of local two phase locking. They are serialized with respect to the transactions of the rest of the system at the point in time of disconnection (this is necessary because they may have read versions of items that existed only at that time). All of the actions of the disconnected site can be considered a single long transaction from the point of view of the rest of the system. Therefore, the algorithm is correct because two phase locking is correct and the distributed database management system enforces global synchronization.

Let us explore in more detail what happens when several sites are disconnecting and reconnecting. If more than one site tries to check-out X at the same time, one of the pseudo-transactions will fail; the item is already locked. Assume that after site S_i disconnects and checks-out X, another site S_k wants to disconnect and check-out Z. Site S_k will observe that a pseudo-transaction from S_i has a write lock on X. Site S_k now asks for and gets data item Z for check-out and disconnects. S_i is unaffected as its transactions involve data item Z as read-only and it reads the version of Z that existed when S_i disconnected.

If S_k reconnects first, it transmits its new value for Z and asks its proxy for any updates that it missed. It will see that X is still write locked by S_i. When S_i reconnects, it will not even know that S_k ever disconnected. Now consider what happens if they reconnect in the reverse order. When S_i reconnects it will update X and release its lock on X by "committing" its pseudo-transaction. In processing the missed updates, it will find out that Z is write locked by a pseudo-transaction from S_k. S_k reconnects, transmits the new value of Z and in processing its missed updates, finds that the pseudo-transaction that locked X has completed.

4.3 Relaxed Check-out mode

In check-out mode, when S_i checks-out an object, the remaining connected sites are prevented from accessing the object which S_i has checked-out. Although this is necessary to preserve serializability, many database systems operate on lesser degrees of isolation [4]. Therefore, we define "relaxed check-out" mode in which the remaining sites can access in browse mode the items that other sites have checked-out. An item can only be checked-out to one site at a time and transactions from sites which have not checked it out are not allowed to wait for a write lock. According to Unland and Schlageter [11], a *browse* lock permits the holder of the lock to read in a dirty mode and does not guarantee consistent reading or permit writing.

The compatability matrix for read, write, and browse locks according to [11] is:

	r	w	b
r	y	n	y
w	n	n	y
b	y	y	y

If S_i disconnects and wants to read and write X, why shouldn't the transactions at the connected sites be able to access the "old" value of X during the disconnect? It is not really necessary to prevent transactions from reading X as the value of X has not been changed yet. Only when the disconnected site which has taken X reconnects and sends out the new value of X will it change the value of X. In the relaxed check-out mode, we allow reads of the old value of X by allowing transactions at other sites to get a browse lock when they cannot get a read lock because the item is write locked by a pseudo-transaction. In this way, other sites can see the old value of X and S_i can read and write X while disconnected and then update the entire distributed database when it reconnects. Relaxed check-out mode allows some non-serializable executions and permits greater concurrency.

Figure 2 shows an example execution using browse locks. In this example, transaction t_5 at site S_k cannot get read locks on X or Y, so it gets browse locks on them. The non-serializable executions occur because of the browse lock. In Figure 2, there is a dependency cycle involving t_1, t_3 and t_5.

Relaxed check-out mode works as follows: We allow a site S_i to disconnect and check-out some items, $I_L(S_i)$ with write locks and also allow S_i to read a set of items $I_F(S_i)$ that were not locked. Other sites can choose to let their transactions get browse locks on $I_L(S_i)$ realizing that they may be reading old data. The rules that must be followed are:

1. Only those items $I_L(S_i)$ locked by pseudo-transactions at disconnect time can be modified by S_i during disconnect.

2. Those items locked by pseudo-transactions at disconnect time can be browsed but not modified by other sites. Any transaction which browses an item in $I_L(S_i)$ knows it has given up serializability. Transactions from other sites attempting to write to those items, $I_L(S_i)$, should be aborted rather than allowing them to wait, since they might have to wait a very long time.

3. Items not locked by pseudo-transactions at disconnect time can be accessed for reading in transactions executed by S_i during disconnect.

This corresponds to degree 2 ANSI isolation levels [4] "cursor stability" if browse locks are only used on write locks held by pseudo-transactions. This is because the browse permits the reading of old data and non-repeatable reads if the item is browsed again after the disconnected site reconnects. However, the read will never be that of an uncommitted and thus truely dirty value. If browse locks are allowed on items that are write locked by any transaction, pseudo or "real", then the appropriate isolation level is degree 1, "browse".

5 Implementation in an Asynchronous System

The implementation in a synchronous system with eager update is fairly obvious as the pseudo-transactions simply obtain locks in a straightforward manner. We therefore give as an example an asynchronous "epidemic" system in which the implementation is less obvious. We use as an example, an epidemic algorithm for replica update that allows update-anywhere capability and maintains one-copy serializability in the system [1, 9].

We use the epidemic style of communication [10, 7, 6], since epidemic communication does not require the distributed system member to be continuously connected as do traditional replica management systems. The members can individually propose updates to the system state and communicate these updates to the other members using epidemic communication. That is, periodically a member decides to contact another member and exchange information. Through this kind of pair-wise communication, the knowledge of the proposed updates spreads throughout the system. Since the members connect for a short time to exchange epidemic messages, then disconnect, this model seems well suited for supporting users in mobile and disconnected environments.

5.1 Epidemic-Based Distributed Database Management

In this approach [1], a transaction, t, executes locally under a local concurrency control mechanism, such as two phase locking. At the point of commit, a *pre-commit* record is written to an *event log*. That record is propagated in an event log to other system members using epidemic communication. As other members become aware of t, they perform the necessary synchronization and communicate their action to each other via epidemic messages.

S_i *pt: wl-X & disconnect *reconnect & $w(X_2)$ c

t_1 $r(X_0)\ r(Z_0)\ w(X_1)$ c

t_2 $r(X_1)\ w(X_2)$ c

S_j *pt: wl-Y & *disconnect *reconnect & $w(Y_4)$ c

t_4 $r(Y_0)w(Y_4)$ c

S_k

t_3 $r(Z_0)\ w(Z_3)$ c

t_5 $b(X_0)b(Y_0)r(Z_3)\ w(Z_5)$ c

Figure 2. Relaxed Check-out Mode Using Browse locks

Each member maintains a vector clock that captures the causal order of events (transaction pre-commits). In addition, each member S_i keeps a 2-dimensional *time table* T_i as defined by Wuu and Bernstein [13], which corresponds to S_i's most recent knowledge of the events at all members. This timing information is included in all epidemic messages. The time table ensures the following *time-table property*: if $T_i[k, j] = v$ then S_i knows that S_k has received the records of all events at S_j up to time v (which is the value of S_j's local clock). The pre-commit record for a transaction from S_i that is stored in the event log, contains the readset, writeset, the values written, and a pre-commit vector timestamp which is the ith row of S_i's time-table, T_i. The timing information is used to determine when a given site is aware of a set of events. Based on the information, transactions can be committed or aborted and records may be garbage collected from the event log.

5.2 Basic Sign-Off

We provide a sign-off and sign-on procedure for the basic sign-off mode for this protocol. Assume that team member S_i wants to disconnect and wants some other team member to act as its proxy during disconnection. S_i, preparing to disconnect, stops generating new transactions. S_i contacts another member S_j to be S_i's proxy. During the disconnection, S_i can only read its local copy of the data. S_i signs-off and disconnects with the following disconnect dialog:

1. S_i contacts S_j, and indicates that it is leaving and requests S_j to act as its proxy. S_i stops accepting epidemic messages from any member other than S_j. S_j stops accepting epidemic messages from any member other than S_i and does not pre-commit any local transactions during the disconnect dialog.

2. S_j sends S_i all of the event log (pre-commit) records that S_j believes S_i has not seen and its timing information. This ensures that S_i knows everything S_j knows before it disconnects.

3. S_i sends S_j its event log records and timing information. This ensures that S_j has an accurate picture of S_i's state so that S_i can be brought back up to date when it reconnects. When S_i has sent this message and knows it has been received, S_i disconnects.

4. When S_j has received and processed this information, it marks its event log with an identifier for S_i to indicate that all records after this point should not be garbage collected since they are needed to bring S_i up to date when it reconnects.

5. S_j resumes sending and receiving epidemic messages and acts as a proxy for S_i.

Transaction processing during the disconnect proceeds as follows: When processing an incoming epidemic record, the proxy S_j checks its event log for conflicts and marks the record aborted if there were conflicts. If the vector timestamps are incomparable, the transactions are considered concurrent and thus their read and write sets must be checked for conflicts. If there were no conflicts, S_j executes the transaction locally as in normal processing. S_j updates its timetable as follows: Let T_j be S_j's timetable and say the update originated at member S_k and has time t_k. Then $T_j[j, k]$ is set to t_k and also $T_j[i, k]$ is set to t_k. This last assignment means that S_i is aware of the record from S_k bearing time t_k. S_i is actually disconnected and unaware of the update, however, S_j has S_i's proxy and is acknowledging updates on behalf of S_i. Other members will believe that S_i knows about their updates and has acknowledged, so the updates can be committed or aborted and garbage collected from their logs. S_j can also commit or abort the transactions, but it must keep the updates in the event log rather than garbage collecting them. S_j finishes processing the incoming record by adding a pre-commit record to its event log thus pre-committing the remote transaction. S_j handles the other records similarly and processes the received timetable information as usual except that it does not garbage collect log records. When S_j pre-commits a local transaction, it increments its local clock $T_j[j, j]$, gives the transaction the timestamp $T_j[j, *]$ and writes a pre-commit record to its log. When S_j has S_i's proxy, it must also change its timetable to indicate that S_i

knows about the transaction by setting $T_j[i,j]$ to the new value of S_j's clock $T_j[j,j]$.

When S_i reconnects, it contacts S_j and gets the updates that it missed, its new log and timetable and retrieves its voting capability. The reconnect dialog is:

1. When S_i reconnects, it contacts the member to which it gave its proxy, S_j, or sends out a query, "Who is acting as S_i?". The member which is acting as S_i will respond and the real S_i can begin the reconnect dialog to retrieve its voting rights.

2. S_j stops processing transactions and incoming epidemic records and sends its entire event log to S_i followed by its timetable, T_j. S_j can now garbage collect its event log and proceed with normal operations as it has given S_i back its proxy.

3. S_i processes the message in the same manner as it would any other epidemic message. S_j's timetable T_j becomes S_i's new timetable. S_i's log and timetable are now up to date and it begins accepting epidemic messages from other members and handling local read-only and update transactions.

Using this procedure, a member can accept proxies for several other members at once. The team member that is acting as a proxy for other team members does not have to maintain separate logs and timetables for each such member. Rather, it must remember which members it is representing and perform a small amount of extra processing. The event log of the member with the proxy can grow large, so it needs to have the option of requiring a complete database copy upon reconnect. This mode is discussed in more detail in [5].

5.3 Check-Out Mode

We now consider the implementation of the check-out mode in the epidemic system. In this case, S_i wants to disconnect and take possession of some data. That is, S_i wants more than a read only copy of the database while it is disconnected. S_i wants to be able to read and write data item X and to be able to read other items such as Y. Other sites will not be able to read or write X.

To accomplish this, we make use of the fact that, according to the epidemic protocol, all transactions attempting to update X will abort if there is a concurrent conflicting transaction. We only need to begin a transaction and acquire the necessary locks, but not complete the transaction while S_i is disconnected. This special type of transaction is a *pseudo-transaction*. Before the disconnect dialog, S_i pre-commits and sends out a record for a pseudo-transaction with X in its writeset with a timestamp containing a distinguished value, say ∞, for S_i's clock. Thus, the timestamp is $T_i[i,*]$ except $T_i[i,i] = \infty$. When sites receive this update record, they process it as they would any epidemic update record by acquiring write locks on behalf of these pseudo-transactions except: any timestamp is incomparable with one containing a ∞ and they do not update their timetable with this value. As a result, this record will remain in their event log and not be garbage collected and will cause the abort of any transaction which reads or updates X (except an update from S_i when S_i reconnects). The database could note which data items are thus restricted and so prevent transactions from attempting to read or update those items before pre-commit. The update record for X that was sent out by S_i will eventually become known by all other sites. When this information gets back to S_i, S_i will know if there were conflicts in which case, the pseudo-transaction indicated by the record failed and must abort. In this case, S_i failed to get the item and can either try again or disconnect without it. If there were no conflicts, the pseudo-transaction was successful and S_i can take possession of the data item X when it disconnects. At this time S_i can begin the disconnect dialog and disconnect with the use of the data items that it successfully acquired.

When S_i reconnects, it sends out an epidemic message with X in its writeset with the current value of X. This record has a normal timestamp. Other sites compare it to records in their log as usual, looking for conflicts. Since its site of origination is the same as the site of the ∞ value in a conflicting record, it is allowed to supersede the log record, thus making it possible to update X again. Thus, when a site S_i disconnects, it can acquire write locks through the use of pseudo-transactions so that those items locked can be updated during disconnection. The pseudo-transactions are completed with a second record upon reconnect that automatically updates the items.

5.4 Relaxed Check-out Mode

As in check-out mode, when S_i wants to disconnect, S_i pre-commits and sends out a pseudo-transaction record with X in its writeset and with a timestamp containing a ∞, for S_i's clock. So, as in check-out mode, the timestamp is $T_i[i,*]$ except $T_i[i,i] = \infty$. When it reconnects, S_i also sends out an epidemic message with X in its writeset. This is recognized by the other sites and X becomes available again.

In relaxed check-out mode, transactions now have the option to acquire browse locks if an item is write locked by a pseudo-transaction. Since a transaction does all of its reads locally, the transaction and its home site can make the decision to acquire a browse lock when it wants to read an item that is write locked by a pseudo-transaction. When the transaction that used the browse capability pre-commits, the browsed item is not included in its readset as that would be detected as potentially causing a non-serializable execution and so the transaction would be aborted. In this way, S_i can acquire write locks and update items during disconnection, while other sites are allowed to browse the checked-out items. The execution in Figure 2 is possible in such a system.

6 Discussion

Mobile computers have added a lot of flexibility to distributed systems. However, distributed databases have traditionally not been very flexible because of the need to synchronize all transactions at all sites. The three proposed modes increase flexibility and will allow distributed databases to take advantage of mobility and to be used in new ways. Basic sign-off mode allows a system which normally requires complete synchronization to continue to function even though a site has disconnected – something that would otherwise be treated as a failure. Our check-out mode allows for independent update of a portion of the database while guaranteeing a smooth reintegration with the distributed database. Relaxed check-out mode increases the concurrency of check-out mode at the cost of perfect transaction isolation, but the degree of isolation is more than sufficient for most applications.

References

[1] D. Agrawal, A. El Abbadi, and R. Steinke. Epidemic Algorithms in Replicated Databases. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 161–172, May 1997.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, Massachusetts, 1987.

[3] M. Faiz and A. Zaslavsky. Database Replica Management Strategies in Multidatabase Systems with Mobile Hosts. In *Proceedings of the 6th International Hong Kong Computer Society Database Workshop*, Mar. 1995.

[4] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufman, 1993.

[5] J. Holliday, D. Agrawal, and A. E. Abbadi. Exploiting Planned Disconnections in Mobile Environments. In *Proceedings of the 10th IEEE Workshop on Research Issues in Data Engineering (RIDE2000)*, pages 25–29, Feb. 2000.

[6] P. Keleher. Decentralized Replicated-Object Protocols. In *Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, Apr. 1999.

[7] K. Petersen, M. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 288–301, 1997.

[8] P. Reiher, J. Heidemann, D. Ratner, G. Skinner, and G. Popek. Resolving File Conflicts in the Ficus File System. In *Proceedings of the Summer USENIX Conference*, pages 183–195, June 1994.

[9] R. C. Steinke. Epidemic Transactions for Replicated Databases. Master's thesis, University of California at Santa Barbara, Department of Computer Science, UCSB, Santa Barbara, CA 93106, 1997.

[10] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, 1995.

[11] R. Unland and G. Schlageter. A Transaction Manager Development Facility for Non Standard Database Systems. In A. K. Elmagarmid, editor, *Database Transaction Models for Advanced Applications*, pages 400–466. Morgan Kaufmann Publishers, 1992.

[12] D. Walborn and P. K. Chrysanthis. Pro-motion: Management of mobile transactions. In *Proceedings of the 11th ACM Symposium on Applied Computing*, 1997.

[13] G. T. Wuu and A. J. Bernstein. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 233–242, Aug. 1984.