Computer Communications 31 (2008) 3209–3222

Contents lists available at ScienceDirect

Computer Communications

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

Timed buffers: A technique for update propagation in nomadic environments

Lorenzo Bettini *

Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy

ARTICLE INFO

Article history:
Received 3 August 2007
Received in revised form 3 May 2008
Accepted 4 May 2008
Available online 13 May 2008

Keywords:
Epidemic propagation
Optimistic replication
Distributed applications

ABSTRACT

Optimistic replication algorithms allow data presented to users to be stale (non-up-to-date) but in a controlled way: they propagate updates in background and allow any replica to be accessed directly most of the time. When the timely propagation of updates to remote distributed replicas is an important issue, it is preferable that a replica gets the same update twice than it does not receive it at all. On the other hand, few assumptions on the topology of the network can be made in a nomadic environment, where connections are likely to change unpredictably. An extreme approach would be to blindly "push" every update to every replica; however, this would lead to a huge waste of bandwidth and of resources. In this paper, we present a novel approach based on *timed buffers*, a technique that tends to reduce the overall number of propagated updates while guaranteeing that every update is delivered to every replica and that the propagation is not delayed.

1. Introduction

The Internet provides means for sharing resources and data among several computers and users distributed over a network, leading to the so-called *global computers* [7]. A user may have several devices that, when connected, can be seen in a uniform way as a single global computer. This scenario can be extended also to multiple users that can exploit connectivity for working in a collaborative shared framework. In this context *Replication* [17,23,37,43] and *Synchronization* [28,18,38,4,31,33] are key concepts, since they enable consistency among data shared by several cooperating devices.

Through replication users can access data even if some sites are not reachable or temporarily non-functional, moreover, the nearest or the idlest site can be used to access shared data in a more efficient way. Synchronization becomes important, not only for keeping all the copies of data, also known as *replicas*, consistent, but also for enabling *off-line operations*. Indeed an network connection may not be available all the time, however, the user can work off-line and, upon reconnection, synchronize his own modified data with a central repository or with other cooperating users.

Traditional replication algorithms such, e.g., those used in distributed database systems [5], quorum consensus algorithms [16] and atomic broadcast protocols [6] usually sacrifice availability of data, since a replica is not accessible until shared data is provably up-to-date. For this reason, these replication algorithms are typically called *pessimistic* [44,19,1,41]: operation requests that might lead to future inconsistencies will be rejected. These are essential in applications where consistency is a major vital concern (such as, e.g., banking transactions). Moreover, the resulting delay, due to blocked access to data during replication, can be accepted in these contexts, where, usually, network is dedicated and reliable, so that failures may not be ascribed to communications.

On the other hand, when the underlying execution context is the Internet, pessimistic algorithms may lead to a huge decrease in execution efficiency and in data availability. Indeed open nets, such as the Internet, are, by their own nature, dynamically evolving structures, since new nodes can get connected or existing nodes can disconnect. Connections and disconnections can be temporary and unexpected. For instance, temporary connections can be established "on the fly" among terminals equipped with wireless devices and ad-hoc paths to services and remote resources can be built dynamically among components. In these scenarios, mobile devices such as laptops, PDAs and cellular phones highly rely on a dynamically evolving communication infrastructure, which is able to reconfigure itself. Thus, the assumption that the underlying communication network will always be available is too strong. Moreover, the knowledge of node addresses may not suffice to establish connections or to perform migrations, since network routes may be affected by restrictions (such as temporary failures or firewall policies).

An alternative to pessimistic replication are *optimistic replication algorithms* [41], which allow data presented to users to be stale (i.e., non-up-to-date) but in a controlled way. Optimistic algorithms propagate updates in background and allow any replica to be accessed directly most of the time. Usenet [22] is probably the oldest Internet service that relies on optimistic replication: indeed it may take some time, even days, for a news group article to reach all the news servers; on the other hand news groups are available

* Tel.: +39 011 6706847.
E-mail address: bettini@dsi.unifi.it

widely all the time. Other popular Internet protocols, such as, e.g., WWW and FTP, use this kind of replication, for caching [46,24,45,2] and mirroring [14] purposes. Optimistic replication is also well suited for directory services such as the widely used DNS [29] and the more recent Active Directory [30]. The key features of optimistic replication algorithms make them more suitable for nomadic applications and mobile computing, and an attractive key enabling solution in wide area distributed applications where communications may be slow and unreliable.

When the timely propagation of updates to remote distributed replicas is an important issue, it is preferable that a replica gets the same update twice than it does not receive it at all. On the other hand, few assumptions on the topology of the network can be made in a nomadic environment, where connections are likely to change unpredictably. An extreme approach would be to blindly "push" every update to every replica but this would lead to a huge waste of bandwidth and of resources.

In this paper, we present a novel approach based on *timed buffers*, a technique that tends to reduce the overall number of propagated updates while guaranteeing that every update is delivered to every replica, without delaying the propagation.

Timed buffers were originally designed to be exploited in middleware platforms for disconnected computing, supporting distributed applications that use replicated shared objects. The original case study was the middleware proposed in [42,25] which allows individual replicas of the shared objects to diverge, with write updates to the replicated objects at each instance being captured. Then, the middleware uses a reconciliation process to update the replicas. However, timed buffers are a general solution that can be adopted in any nomadic environment.

Platforms like the one just mentioned require the timely propagation of updates among replicas. The underlying network topology assumptions are few: the platform must be able to support adhoc wireless nodes, high-latency wide area networks (WAN), and low-latency local area networks (LAN). A possible use of this platform is, for instance, the implementation of a collaborative design tool, where the drawing is the shared object. Many designers in the same office can be working and altering the drawing; at the same time a consultancy firm on a different continent can also be altering the drawing, and two designers can be on a plane where they are both altering the drawing as well. In this case, the designers in the office have LAN connectivity, the consultancy firm is connected, but via a high-latency link, and the designers on the plane are connected to each other, but isolated from the rest of the users of the shared data. Furthermore, over the lifetime of a single application instance, multiple different connectivity topologies of the nodes are possible. The platform embodies the concept of tentative views of the data, and, as such, the sooner an update on one replica is propagated to the other replicas, the sooner the tentative views of the data are updated. This, in turn, minimizes the number of conflicts during reconciliation. On every node, a daemon is run to provide the platform services, and the daemon maintains a list of other nodes with which it believes it can currently communicate. In order to achieve this, an explicit node discovery protocol is used, which involves the use of multicast beacons. When a node receives a beacon, it replies with any necessary information, such as IP address etc. In this paper, we will concentrate only on the algorithms and protocols used to control the propagation.

A general survey of existing systems and basic *optimistic replication algorithms* can be found in [41], where a classification schema of update propagation in distributed systems using optimistic replication is presented. This classification schema is based on single or multi-master, log- or content-based and whether the updates are propagated in a push or pull manner. If a particular node maintains a master copy of the shared data, then the system is described as *single master*, otherwise it is *multi-master*. If the system maintains a log of updates applied to the replicated data, rather than comparing the state to determine differences, the system is *log-based* as opposed to *content-based*. Finally, if the nodes request updates from other replicas then the system is *pull-based*, rather than *push-based* (a node pushes updates to other nodes). According to this classification, the platform we are considering is a *multi-master*, *push-based* and *log-based* platform.

After a discussion about some related works, the paper proceeds as follows: in Section 2 we recall the standard techniques for multi-master, push-based update propagation and in Section 3 we present our novel approach for optimizing epidemic update propagation. Section 4 presents the protocol for establishing a communication between two nodes, in order to exploit our technique. Some experimental results are presented in Section 5. Section 6 concludes the paper.

1.1. Related work

Many systems use propagation algorithms in distributed or mobile environments. Bayou [12] provides the application writers with a replicated, weakly consistent, data storage engine. There is no centralized data store and it relies on pair-wise communications between computers; every replica eventually receives updates from all the other replicas through a chain of pair-wise propagation of data, using anti-entropy, performed automatically at a set interval, or when requested by an application. In our approach, updates have to be propagated as soon as possible, and it is not simply pair-wise.

Ficus [37] is a distributed file system that allows files, grouped in *volumes*, to be selectively replicated without storing the entire volume. It maintains data consistency with *update notification messages* and *reconciliation*. In our platform, all nodes directly propagate the updates rather than propagating a notification message. However, an optimization aimed at reducing the size of messages when the size of updates is huge, is sketched in Section 6. *Rumor* [38] is an intellectual descendant of Ficus, and it is a peer-to-peer reconciliation-based replication service aimed at distributed file systems. It uses *version vectors* in order to guarantee that each update is issued with a unique signature and that the same update is never transmitted to the same replica more than once. In [36], an effort is made in order to manage these vectors in an efficient way and save space for their storage. The idea is that once all replicas have the same value for an update issued by a replica R, then R's entry can be removed from the vector. Obviously, compression requires consensus, in order to be performed in a safe way.

In [21], another solution (*hierarchical matrix timestamps*) is adopted in order to reduce the size of timestamp matrices. The sites that store a replica are partitioned among a set of *domains*; each site stores two timestamp matrices: one for the information about replicas in the same domain, and the other matrix contains summary information about the other domains. Entries in this second matrix contain the minimum of the timestamps of all the replicas in another domain. So the estimation is less accurate but it is still safe. With this approach if these nodes are split in $O(\sqrt{n})$ domains, the timestamp matrices can require only a linear space, instead of quadratic. In our approach, the dimension of the timestamp matrices can be split in two (as sketched in Section 6) so that only the most used part is kept in memory. Moreover, our assumption of a nomadic environment is not well suited for a static division of nodes in domains.

Roam [34], built using the *Ward* architecture (*Wide Area Replication Domain*) [35] is a replication system redesigned specifically for mobile computing. The Ward model combines peer-to-peer and client–server providing domains of grouped nodes and also handles replicas' domain crossing. The *ward master* is the only link to the other wards. In a highly mobile system, like the ones we are addressing, domain crossing could be very frequent, and these situations should be handled implicitly; indeed in our system we do not keep groups (we have a real multi-master approach), and the nodes adapt themselves to the topology of connections by exchanging information.

Porcupine [39,40] is a cluster-based mail server. Its architecture is fully dynamic in that every node can manage any user's profile and store any user's e-mail messages. A set of nodes is chosen on which to replicate and store a message based on the load of the node and message affinity. Any replica can issue an update at any time, and the update is then propagated by pushing the new object state to others in background (*Thomas write rule* [44] is used to resolve conflicts between updates). When the issuer of an update knows that all the target nodes received an update (by receiving *ack* messages), the update is *retired*, and a retire message is sent to the target nodes. Basically our schema for propagation using timed buffers (Section 3) is similar to the retire algorithm in Porcupine but the updates are not removed. Moreover we do not transmit the state of the object, but only the actions in order to bring the state up-to-date (log-based), and conflicts between these actions can be handled, separately, by a reconciliation module.

Deno [8,10] is an object replication system, specifically designed for use in mobile and weakly connected environments. It uses *weighted voting* [9] for selecting transactions that have to be committed or aborted. Clients connect to a peer server, which communicates through pair-wise information exchanges, so only servers take part in voting and in synchronization, but there is no primary server that owns an item. Election information flows through anti-entropy sessions in a uni-directional way. Deno supports both pushing and pulling of updates, but uses pulling by default.

2. Update propagation

A key concept in log-based replication is the propagation of updates to replicas. In this section, we recall the standard techniques for multi-master, push-based update propagation, and introduce the basic concepts that will be used in our approach. Hereafter, we will use indistinguishably the term *node* and *replica* to refer to the application storing and modifying the shared data.

A simple and safe way of performing propagation is using *blind pushing*: a replica propagates the update indiscriminately to every replica it can communicate with. This may obviously lead to flooding and generate many duplicate information, if the replicas share many connections with other replicas: most of them will end up receiving the same update by many replicas. Indeed a pull-based propagation would not suffer from this problem: replicas never receive the same update twice, because the set of updates to receive is determined by the replica itself by polling other replicas. On the other hand, push-based propagation systems are more efficient since they do not require polling and they reduce the update propagation delay: the update is pushed (propagated) right after the issuance.

In order to deal with duplicate updates, replicas have to filter the incoming updates, so they do not handle the same update more than once. One simple way to filter these updates is to use *Timestamps*,[1] i.e., every update is added a unique timestamp by the replica that originated it; then if an update is received with a timestamp that has already been handled, it is discarded. In our implementation a timestamp is simply a number that increases monotonically, i.e., a counter that is incremented at each update issuance, and it is part of the identifier of an update: an update is uniquely identified within the system by its timestamp and the issuer replica that originated it.

If we want to avoid blind pushing, some sort of knowledge of the state of the other replicas has to be kept. Obviously, we can only estimate the state of the other replicas, as we do not know the complete configuration and topology of the network: it may be continuously changing since we are dealing with mobile applications. We use *timestamp matrices*, which are an example of state-estimating technique, used in multi-master, log-transfer, push-based systems [27,47,17]. The main idea is that each replica estimates the state of the other replicas it is connected to, and propagates to each of them only the updates that are likely to miss.

This way of performing propagation is also known as *epidemic propagation* [17,11,15,20,13], since the update is very similar to a disease that infects the whole population (the other replicas).[2] The user performs operations on its own data and, asynchronously, a separate activity (also known as *anti-entropy*) compares the estimated state of other replicas and propagate updates to replicas that store older versions. Anti-entropy can also be used as a symmetric mechanism: it allows two replicas to bring each other up-to-date.

Every replica stores a timestamp matrix indexed by node identifiers (e.g., node guids). By TM_i we mean the timestamp matrix of node i. The entries in the timestamp matrix are timestamps of updates, and they have the following meaning for a node i:

- $TM_i[i]$ is the *timestamp vector* that summarizes the state of the node i itself, i.e., the most recent timestamps of received updates issued by each replica. Thus, $TM_i[i][j] = n$ means that the last update created by j, that i received, has timestamp n.
- The other rows in the matrix show an estimate of the timestamp vectors of other replicas, thus $TM_i[j][k] = n$ means that i knows that the last update created by k, that j received, has timestamp n. j might have received other updates from k but that knowledge has not yet reached i. This estimate is said to be *conservative*: it cannot happen that the last update that j actually received from k has timestamp m with $m < n$.

It is important to note that we are considering the *issuer* of the update (i.e., the real creator of the update), and not the *sender*: in fact due to the epidemic propagation and to the network topology, a replica could receive an update by someone else different from the issuer. The creator of the update is stored in the update itself (as previously stated, it is part of the update's identifier), while the sender is not.

During the propagation of an update with timestamp n issued by k, i will send it to j only if $TM_i[j][k] < n$. The node i will not update that entry in its timestamp matrix until it receives an acknowledge from the receiver. Indeed every sent update has to be acknowledged by the receiver, by means of an *ack* message.

Timestamp matrices are *piggybacked* in messages (also when sending an ack for a received update). Thus, when i sends a message to j they will both have the timestamp matrix of each other. Then they both perform a *merging operation* on the timestamp matrices. This operation consists of computing the pair-wise maximum for every element. During the merging, the estimate of the state of the other replicas is updated with the estimate contained in the received timestamp matrix. Obviously, a node updates its own timestamp vector only upon receiving an update, not during the merging. Note that when a node receives a timestamp matrix attached to a message, this matrix may be different in size from

[1] A timestamp could be any number that increases monotonically, such as a logical clock [26], a wall clock [28], or any simple increasing counter. Timestamps are also used when the contents, instead of the logs, are transferred; for instance the *Thomas write rule* [44] associates with each replica a timestamp, that records the last time the replica was modified. New contents are downloaded only when they are newer according to the timestamp.

[2] Epidemic algorithms, apart from the metaphor with real diseases, also benefit from the results of mathematical theories of epidemics [3] that, in case of a push-based approach, show that the entire population is eventually "infected" in expected time proportional to the log of the entire population [32].

its own timestamp matrix, since the two nodes (the sender and the receiver) may be connected to different nodes. During the merging operation the information about these new nodes, to which there is no connection, can be discarded, or, alternatively, can be exploited for trying to establish a connection also with these nodes.

In the rest of this section, we show the basic procedures and data structures for simple epidemic update propagation. It is important to observe that, in our context, when we use the term "connection" between two nodes, we do not necessarily mean that a real network connection is established between them, but only that the two nodes have previously established a connection. Due to the high dynamism of the nomadic scenario that we are considering, such a connection may be closed without any notice.

In the algorithms when a procedure refers to TM_{me}, it refers to the timestamp matrix of the node it is executed on. The procedure `SendUpdate(n,u)` sends the update `u` to the node `n`, and `SendAck(n,u)` sends an ack about the update `u` to `n`. The timestamp matrix of the sender node is automatically and implicitly attached to messages before they are actually delivered.

We assume that an update has the following structure:

```
Update:
    id: integer // the timestamp
    action: Action // the actual content of the update
    issuer: UpdateIssuer
```

Upon receiving an update a node checks whether it has already received that update and if it has, it simply discards it; otherwise it uses it (e.g., for reconciliation), it updates its own timestamp vector in its timestamp matrix and it starts the epidemic propagation. In any case it notifies the sender that it received the update with an ack message.

```
proc ReceiveUpdate(u: Update, from: node)
    SendAck(from, u)
    if $TM_{me}[me][u.issuer] < u.id$ then
        // accept update
        $TM_{me}[me][u.issuer] := u.id$
        // handle the update and propagate it
    else
        // discard update
    endif
```

The epidemic propagation takes place according to the contents of the timestamp matrix. The platform maintains a set *connected* that contains the identifiers of nodes with which it is connected:

```
proc PropagateUpdate(u: Update, from: node)
    for every node i in connected where $i \neq from$ and $i \neq u.issuer$
    do
        if $TM_{me}[i][u.issuer] < u.id$ then
            SendUpdate(i, u)
        endif
    enddo
```

Thus, an update is propagated to a node *i* only if it is estimated that *i* has not received that update yet (and, obviously, if it is neither the issuer nor the sender of that update).

Note that the timestamp matrix is only an estimate of the state of other replicas, so it will be very likely, in a highly connected graph that a node will receive the same update more than once; but on the other side, it will also receive the update as soon as possible from the fastest connection.

Let us consider a scenario where a node *a* produces an update, it will deliver it to all the nodes it is connected to, say *b*, *c* and *d*; as soon as *b* receives such an update it will deliver it also to *c* and *d*. The same happens for *c* and *d*. Let us analyze the best and the worst case, assuming that *a* will send the update to every other nodes in a network where every node is connected to every other node $(n-1)$:

- In the best case (depicted in Fig. 1) the first node that will receive the update, say *b*, will propagate it to everyone but the issuer itself $(n-2)$; note that at the moment *b* cannot know that the others have received it. *c* may receive the update from *b* sooner than from *a*, and so it will only propagate it to *d*, which, if it had not yet received the update from *a* it will have not to propagate it to anyone else (in fact by the piggybacked matrix from *c* it knows that *b* already received the update). Thus, generalizing this scenario to *n* nodes, the total number of messages that travel in the net is: $(n-1)+(n-2)+\cdots+2$, and thus $O(n^2)$.
- In the worst case (depicted in Fig. 2) all the $n-1$ nodes will end up receiving $n-1$ messages, and thus it is still $O(n^2)$.

Actually there would be no need for the other nodes (*b*, *c* and *d*) to propagate the update, because the issuer of the update itself, *a*, will deliver it or has already delivered it (in fact we are considering a fully connected graph). However, we want to make sure that every replica will eventually get the update, and we are also considering disconnections; thus if *d* disconnects from *a* before the update is sent, *d* might not receive the update if *b* and *c* (or at least one of them) did not propagate it.

Summarizing, it is safe to let the epidemic propagation to take place, but, at the same time, it seems to be feasible to reduce the number of messages spread through the network. In the following sections, we will propose some optimizations for reducing the number of messages exchanged during the epidemic propagation.

3. *Timed buffers*: optimizing epidemic update propagation

In this section, we present our novel approach for reducing the number of messages during epidemic update propagation. All the updates will be guaranteed to be sent to each replica and no delay will take place in ordinary conditions; this means that if a node is effectively connected to some nodes that have an update, it will receive that update without any delay (with respect to the simple algorithms presented in Section 2).

Central in our approach is the concept of *time-out*: this is used for specifying the time the replicas will have to wait before starting the epidemic propagation; in the meantime the update is kept in a buffer with an associated time-out: a *timed buffer*. Let us briefly sketch our technique: nodes should wait some time before starting the propagation; if the time-out expires, the propagation actually takes place. This wait will hopefully be useful for avoiding sending an update to a replica that is estimated to have already received it from another replica. However, this wait should not delay the delivery to other replicas. In order to do this, when a replica is propagating an update to another replica, it should also specify the other replicas it is going to send the update to.

The two variants of the timed buffer technique, presented in the next sections, differ because of the place where the timed buffer is handled.

3.1. Receiver-based timed buffers

The approach we are going to present in this section is called *receiver-based* because the buffer is kept on the receiver node. Basically the nodes involved in epidemic propagation of an update perform these steps:

- The sender sends an update together with a time-out to all the replicas it is connected to.

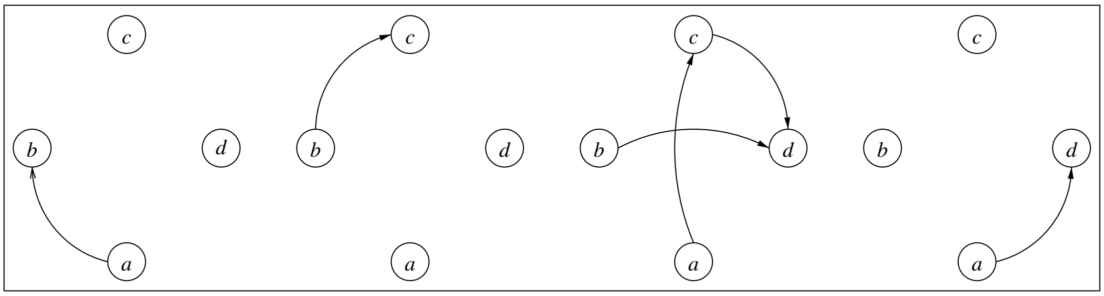


Fig. 1. Four moments in succession of a best case situation (all nodes are connected).

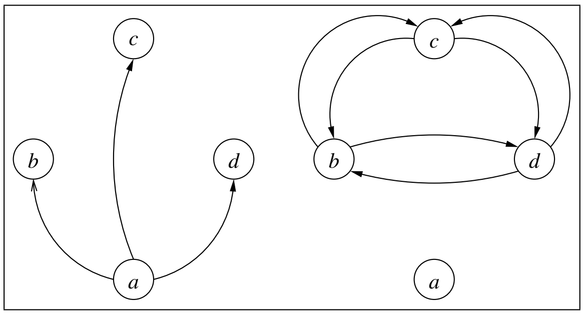


Fig. 2. Two moments in succession of a worst case situation (all nodes are connected).

- If it is acknowledged by them, it sends another message (a *cancel* message) to the other replicas, specifying that the update is no longer to be propagated. This notification message is sent only if the time-out is estimated not to be already expired on the other nodes.
- If no cancel message is received within the time-out, the other nodes will propagate the update.

We call this sort of transaction a *time-out session*.

While it may seem that many messages are sent also in this case, let us analyze the best and the worst case:

- In the worst case, all time-outs will expire, and thus we are in the same situation as the simple algorithm presented in Section 2: $O(n^2)$.
- In the best case, i.e., all nodes receive the update from the issuer and acknowledge the update before the time-outs expire, the messages that travel in the net will be:
  1. A message with the original update sent by the issuer to every node.
  2. An ack sent by every receiving node.
  3. A cancel message sent to every node.

  So the issuer of the update will send $2(n-1)$ and receive $(n-1)$ messages, and thus $O(n)$ messages travel in the net.

Thus, we can only get better using this approach.

Note that, if we are considering clusters of fully connected nodes with fast and reliable connections, by using this timed buffer approach, the number of messages will decrease (linear), while the simple epidemic propagation would still be quadratic. Indeed, in such a situation, the simple approach, presented in Section 2, would probably experience the worst case almost always, as, since the connections are fast, all nodes will receive the updates almost simultaneously, and they would propagate the updates to every other nodes.

This solution, however, could delay propagation to other nodes that are not related to the time-out session: in fact, if one of these nodes, say *b* is also connected to a node *e*, to which *a* is not connected, the propagation to *e* should not be stopped by the time-out, because *b* could send the update to *e* meanwhile (Fig. 3). Thus, when a node sends an update to a node *n* with a time-out, it should also specify the nodes that are involved in the time-out session: the nodes in the connection set that are common to the sender of the update and to *n*. In our example, the nodes that belong to this set for *a* and *b* are *c* and *d*, but not *e*. These sets can be estimated by means of some additional information that can be transmitted together with the timestamp matrix, and stored by a replica together with its own timestamp matrix.

Let us consider the two cases when such an estimation is wrong and show that, however, the estimation is still conservative:

- If the sender thinks that one of the receivers, say *b*, is not connected to one of the other receivers, say *c* (while *b* is actually connected to *c*), then *b* will soon propagate the update to *c*.
- If the sender estimates that *b* is connected to *c*, but this is not true, *b* could not send the update to *c* anyway.

Thus, once again, no update is either delayed or lost. Note that a node is not estimating the entire network topology, but only the connections of the nodes it is connected to. This estimate is kept up-to-date through the additional information received together with messages and processed during the merging of timestamp matrices. Let us recall that the additional information, such as timestamp matrices and connection sets, are always attached by the sender to every kind of messages that is exchanged in the following propagation algorithms.

Hereafter, in the procedures that we are going to present we will use the following structures, available in every replica: *connected* is the set of nodes (node identifiers) to which the replica is connected to (up to the notion of connectivity introduced previously); *connections* is the estimate of other nodes' connections, thus *connections*[*i*] stores the connections of node *i* as estimated by the replica.

We assume that an `Hashtable` data type is available that supplies operations for inserting (`put`) a pair (key, value), for retrieving

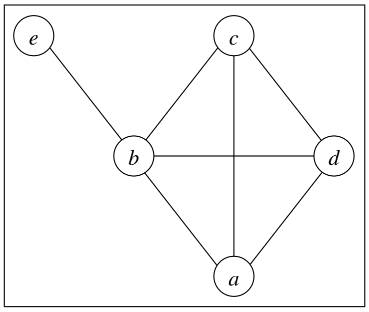


Fig. 3. *a* and *b* do not have *e* in their common connection set.

a *value* given a *key* (get), and for extracting a value given a key (extract). When the value is a set (e.g., of nodes) we assume that a put will not overwrite the previous value, but simply add new elements to the set. Records are shown in angle brackets ⟨ ⟩. For the sake of simplicity, when we refer to "node *i*" we mean "node with identifier *i*". Finally the procedure copy returns a copy of the argument.

The procedures for the propagation will make use of and update some tables (altogether these tables implement timed buffers):

```
TOSessionTable: Hashtable
  key: ⟨ Update, node ⟩
  value: ⟨ StartingTime, to_node_set, Timeout ⟩
```

where StartingTime is the time the update was sent (the time-out should be considered from this time on), to_node_set is the set of the nodes that are involved in this time-out session, and Timeout is the time-out associated with this set.

```
retirement_table: Hashtable
  key: ⟨ Update, node ⟩
  value: retirement_set (set of nodes)
```

This table stores the nodes (retirement_set) that are involved in the time-out session for a specific update sent by a specific node. We assume that we can use a system function, getTime, in order to retrieve the current time.

Given a set of nodes, the following procedure will take care of filtering out the nodes that the update is not to be propagated to:

```
proc FilterOutNodes(u: Update, nodes: set of nodes)
    for every node i in nodes do
        if $TM_{me}[i][u.issuer] \geqslant u.id$ or
            u has already been sent to i or
            i has an associated time-out for u then
        remove i from nodes
        endif
    enddo
```

Note that the first condition implies the second one, but the contrary does not hold: indeed an update may have been sent to a node, but that node has not sent the ack yet; in that case $TM_{me}[n][u.issuer] < u.id$; but the update should not be sent another time. The second condition can be easily handled by storing in a table, say *sent*, the nodes an update *u* has been sent to. When the sender receives an ack back for that update from a node *n*, this node is removed from the list associated with *u* in the table *sent*; the entry for *u* will be deleted from this table when there are no more nodes associated to *u*. Finally, the last condition is tested by inspecting the TOtable (shown later).

Upon creation of an update u the following procedure is called

```
proc PropagateCreatedUpdate(u: Update)
  var nodes_to_propagate :=copy(connected)
  FilterOutNodes(u, nodes_to_propagate)
  PropagateUpdateWithTO(u, nodes_to_propagate)
```

and the procedure for the propagation follows:

```
proc PropagateUpdateWithTO(u: Update, n: set of nodes)
    for every node i in n do
        var to_node_set :=connections[i] ∩ connected
        for every node m in to_node_set do
            retirement_table.put(⟨ u, m ⟩, i)
        enddo
        TOSessionTable.put(⟨ u, i ⟩,
            ⟨ getTime(), to_node_set, TIMEOUT ⟩)
        SendUpdate(i, u, to_node_set, TIMEOUT)
    enddo
```

When a node *n* sends an update to a group of nodes, for every node *m* in this group, *n* estimates the nodes that are connected both to *m* and itself, the set to_node_set. For the nodes in this set, that will be delivered together with the update, the receiver node will store the update in a timed buffer, but it can propagate the update to any other node that is not in this set.

TIMEOUT could be a prefixed value, or a dynamic value that is computed on the fly, e.g., according to the latency of getting acknowledgments back from the receivers: it could be the maximum of all the average latencies for all the nodes the update is sent to. Alternatively, it could be different for each single node.

The next procedures handle an ack from node from about the update u.

```
proc ReceiveAck(from: node, u: Update)
    retirement_set :=retirement_table.extract(⟨ u, from ⟩)
    for every node n in retirement_set do
        CheckRetirement(n, from, u)
    enddo

proc CheckRetirement(n, from: node, u: Update)
    ⟨ StartingTime, to_node_set, Timeout ⟩ :=
    TOSessionTable.get(⟨ u, n ⟩)
    to_node_set.extract(from)
    if to_node_set.isEmpty() then
        if (getTime() - StartingTime) < (Timeout - $\delta$) then
            SendCancel(n, u)
        endif
        TOSessionTable.extract(⟨ u, n ⟩)
    endif
```

So every time an ack from a node *n* about an update *u* is received, *n* is removed from the time-out set of all the other nodes. If such a set becomes empty for a node *m*, a cancel message is sent to *m*. Such a cancel message is sent only if the time-out on the remote node is estimated not to be expired already. The δ value should serve as a value that takes these additional factors into account.

These are the procedures that nodes execute when they receive an update with a time-out. They use a table indexed by update identifiers.

```
TOtable: Hashtable
  key: Update
  value: ⟨ timeout_node_set,
    StartingTime, Timeout, TimeoutProc ⟩

proc ReceiveUpdateWithTO(u: Update,
   to_node_set: set of nodes, Timeout: int)
  var deliver_set :=connected-to_node_set
  TOtable.put(u, ⟨ to_node_set, getTime(),
    Timeout, TimeoutProc(u) ⟩)
  FilterOutNodes(u, deliver_set)
  PropagateUpdateWithTO(u, deliver_set)
```

The update can be immediately propagated to the nodes (deliver_set) that are not in the to_node_set. Note that the procedure FilterOutNodes is still used. For the other nodes a timed buffer is set, and a procedure (TimeoutProc) will be triggered if that time-out expires before receiving the cancel message. This procedure will start the update propagation for the nodes in to_node_set, after further filtering them.

```
proc TimeoutProc(u: Update)
  ⟨ to_nodes, _, _, _ ⟩ :=TOtable.extract(u)
```

```
    FilterOutNodes(u, to_nodes)
    PropagateUpdateWithTO(u, to_nodes)
```

The values for _ fields are simply ignored. Otherwise, upon receiving a cancel message, the timed buffer and the associated time-out is canceled:

```
proc ReceiveCancel(u: Update)
  TOtable.extract(u)
```

We conclude with a possible scenario: in Fig. 4 an example of a session is depicted, together with the `TOSessionTable` and `retirement_table` of the sender, the node *a*; in the messages that are sent to the nodes, apart from the update, also the `to_node_set` is shown. Note that `retirement_table` and `TOSessionTable` have the same set of nodes: this makes sense, since the connections are symmetric. We are keeping separate tables in order to keep the algorithms simple in this place. In this example a different time-out is sent to each node.

Let us now examine a possible execution in the scenario of Fig. 4. Since the sender is *a* we consider the state of its tables while they are evolving. The initial tables are

TOSessionTable:	retirement_table:
⟨ u,b ⟩, ⟨ t_b, [c,d] ⟩	⟨ ⟨ u,b ⟩, [c,d] ⟩
⟨ u,c ⟩, ⟨ t_c, [b,d] ⟩	⟨ ⟨ u,c ⟩, [b,d] ⟩
⟨ u,d ⟩, ⟨ t_d, [b,c,e] ⟩	⟨ ⟨ u,d ⟩, [b,c,e] ⟩
⟨ u,e ⟩, ⟨ t_e, [d] ⟩	⟨ ⟨ u,e ⟩, [d] ⟩

If *a* receives an ack from *b* the tables become:

TOSessionTable:	retirement_table:
⟨ u,b ⟩, ⟨ t_b, [c,d] ⟩	
⟨ u,c ⟩, ⟨ t_c, [d] ⟩	⟨ ⟨ u,c ⟩, [b,d] ⟩
⟨ u,d ⟩, ⟨ t_d, [c,e] ⟩	⟨ ⟨ u,d ⟩, [b,c,e] ⟩
⟨ u,e ⟩, ⟨ t_e, [d] ⟩	⟨ ⟨ u,e ⟩, [d] ⟩

If now *d* sends an ack, then *a* can send a *cancel* message both to *c* and *e*, in fact their sets in the `TOSessionTable` have just become empty (and their entries will be deleted as well):

TOSessionTable:	retirement_table:
⟨ u,b ⟩, ⟨ t_b, [c] ⟩	⟨ ⟨ u,c ⟩, [b,d] ⟩
⟨ u,d ⟩, ⟨ t_d, [c,e] ⟩	⟨ ⟨ u,e ⟩, [d] ⟩

Then, when also *c* sends an ack, a cancel message can be sent to *b*, and so on.

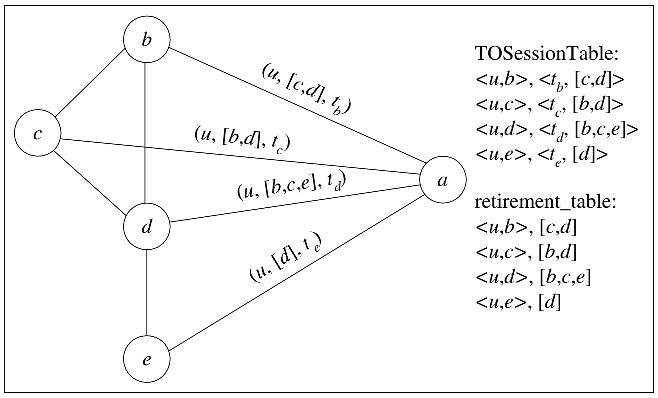


Fig. 4. Receiver-based timed buffers: an update propagation.

3.2. Sender-based timed buffers

In the sender-based approach the sender node still specifies, in the message, the set of nodes to which the update must not be immediately propagated, but, instead of specifying a time-out, a timed buffer is set locally: if an ack from a node is not received within that time-out a *propagate* message is delivered to the other nodes, which then propagate the update to that node. Otherwise the nodes that receive an update do not start propagation by default. In other words, the epidemic propagation is automatically executed only by the issuer of the update; as regards the other nodes, that receive an update, they execute epidemic propagation only towards the nodes that are not specified in the message, while for the remaining nodes, it is executed only on demand.

Using this approach a message is saved if no time-out expires (the *cancel* message in receiver-based approach), and thus the number of messages is still O(n), but with a smaller factor ($2(n-1)$ instead of $3(n-1)$). However, if the time-out should expire the sender will specify, in the *propagate* message, the set of nodes that have not sent the ack: the propagation would then be confined to those nodes only, thus the number of messages would be quadratic in the number of nodes that did not ack, which is likely to be small.

The procedure to filter out nodes is smaller because the time-out is associated to the nodes the update is sent to, and that have not acknowledged yet: these are the nodes for whom the second test fails.

```
proc FilterOutNodes(u: Update, nodes: set of nodes)
  for every node n in nodes do
    if $TM_{me}[n][u.issuer] \geqslant u.id$ or
      u has already been sent to n then
        remove n from nodes
    endif
  enddo
```

and the procedures for the propagation are

```
proc PropagateCreatedUpdate(u: Update)
  nodes_to_propagate :=copy(connected)
  FilterOutNodes(u, nodes_to_propagate)
  PropagateUpdateWithTO(u, nodes_to_propagate)

proc PropagateUpdateWithTO(u: Update, nodes: set of node)
  for every node n in nodes do
    not_to_send_to :=connections[n] ∩ connected
    SendUpdate(n, u, not_to_send_to)
  enddo
  StartTimeOut(TIMEOUT, TimeoutProc(u))
```

`StartTimeOut` sets a time-out and a callback procedure to be triggered when the time-out expires:

```
proc TimeoutProc(u: Update)
  var to_set :=nodes that did not acknowledge u
  for every node m in connected do
    propagate_to :=to_set ∩ connections[m]
    if not propagate_to.isEmpty() then
      SendPropagate(m, propagate_to, u)
    endif
  enddo
```

Thus, when such a time-out expires, the nodes that have not acknowledged the update *u* are collected, and *propagate* messages are sent to the nodes that are connected to some of the former. Upon receiving a *propagate* message, the update will be propagated to the specified nodes, after filtering nodes out with the same procedure.

```
proc ReceivePropagate($u$: Update, propagate_to: set of node)
    propagate_to :=propagate_to ∩ connected
    FilterOutNodes(propagate_to)
    PropagateUpdateWithTO($u$, propagate_to)
```

Note that the first operation is necessary because the sender only estimates the connections of the receiver. Moreover, the filter procedure is still applied, since a node that receives a *propagate* message may know that some of the nodes to propagate to already have the update.

Thus, the sender-based approach performs even better, but it has a drawback: if a node *b* receives an update from *a*, where it is specified that it must not be propagated to *c*, and *a* disconnects or fails, *b* has no way of knowing if the last updates received from *a* have actually reached all the destinations; if *b* should sense that *a* has disconnected, then it should propagate all the updates from *a* that are not known to *c* (according to the timestamp matrix). If these updates are many, a resynchronization session (Section 4) with *c* could be performed instead.

4. Starting a connection

The propagation algorithms presented in Section 3 apply to replicas that are already connected and communicate. Another anti-entropy synchronization has to be executed when two nodes connect to each other. This will lead them both to the right starting state from which the propagation algorithms based on timed buffers can be safely executed.

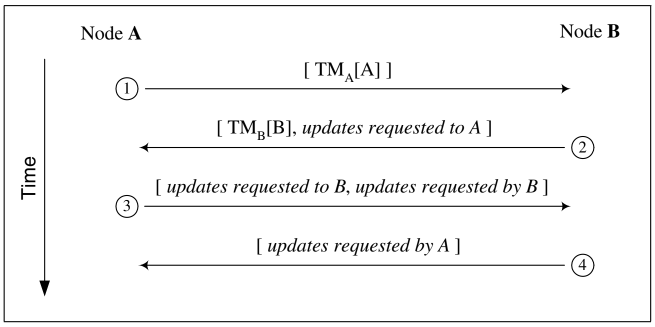
When a node *A* starts to communicate with another node *B*, they both execute a protocol that allows them to bring each other up-to-date. Basically the two nodes exchange their timestamp vector, and then they request possible missing updates. Thus, *pull-based* resynchronization is performed upon connection with another node. Fig. 5 shows a session of this connection protocol between *A* and *B*.

The following steps are performed by a node *A* that starts to communicate with a node *B*:

1. *A* sends its own timestamp vector.
2. In response, *A* receives the timestamp vector (`TV`) of *B* together with the list of updates that *B* is requesting (`update_request`).
3. *A* updates its timestamp matrix according to this vector, it computes the missing updates (the ones that *A* does not have, but *B* does) and sends a request for these updates to *B* together with the updates that *B* had requested.
4. Finally *A* receives from *B* the updates it had requested with the previous message.

The procedures that implement these steps follow:

```
protocol UponConnectionWith($n$: node)
    SendTo($n$, $TM_{me}[me]$)
    ⟨$TV$, update_request ⟩ :=ReceiveFrom($n$)
    update $TM_{me}[n]$ with $TV$
    missing_updates :=Diff($TV$, $TM_{me}[n]$)
    SendTo($n$, ⟨ missing_updates, updates requested by $n$⟩)
    requested_updates :=ReceiveFrom($n$)
```



Node **A**
Node **B**
Time
1
[$TM_A[A]$]
[$TM_B[B]$, *updates requested to A*]
2
3
[*updates requested to B, updates requested by B*]
[*updates requested by A*]
4

Fig. 5. Messages exchanged during the protocol.

The procedure `Diff(V1,V2)` computes the difference between two timestamp vectors; note that the two vectors may not have the same length. Indeed if it is the first time that *A* communicates with *B*, $TM_A[B]$ will not even be defined, neither will $TM_A[A][n]$ for some *n* to which *B* is connected to but *A* is not. Such undefined entries are assumed as 0 in the `Diff` procedure, and they are automatically created during the update of TM_A with the vector received by *B*.

On the other node, *B* in this example, there will be a process that waits for a connection request; when such a request arrives, the sender of the request and its timestamp vector (respectively, `from` and `TV` in the following procedure), are retrieved and the complementary steps are executed:

```
protocol UponReceivingConnection
    while true do
        ⟨ $from$, $TV$ ⟩ :=Wait4ConnectionRequest();
        update $TM_{me}[from]$ with $TV$
        missing_updates :=Diff($TV$, $TM_{me}[from]$)
        SendTo($from$, ⟨ $TM_{me}[me]$, missing_updates ⟩)
        ⟨ update_request, requested_updates ⟩ :=
          ReceiveFrom($from$)
        SendTo($from$, updates requested by $from$)
    enddo
```

Should an update be received by *A* during this protocol it would not be missed by *B*:

- If such a new update is received before time 1 (refer to Fig. 5), it will be included in the vector that *A* sends to *B*, and thus it will be requested by *B*, if *B* does not have it already.
- If it is received after time 1, but before time 2, then *A* may not know anything about the state of *B*; in this case, since *A* wants to stay conservative, it will propagate that update to *B*.
- In all the other cases, *A* already has an estimate of the state of *B* so it will decide whether that update should be propagated or not.

5. Experimental results and analysis

In order to test these algorithms we built a simulator that simulates the propagation of one update in a graph with a randomly generated topology. The simulator consists of a Java program that implements timed buffers and the propagation techniques described in the paper (also the simple push and pull approaches); then the simulation is performed on a single machine, using discrete time (discrete-event simulation). The connections are generated according to a certain percentage: when we specify, for instance, that the percentage of connections is 20%, we mean that every node is connected to 20% of the other nodes. The latency of communications is set to 10 ms. There is no mobility in these simulations, and no disconnections either. We are considering the number of messages that are sent in such graphs in order to propagate the update to every node and the time it takes for every node to receive the update.

5.1. Number of messages

In Figs. 6 and 7, we let the number of nodes vary, we generate connections with a percentage of 20% and 50%, respectively, and we count the number of messages sent in the graph. We note that, as expected, the two proposed optimizations reduce the number of

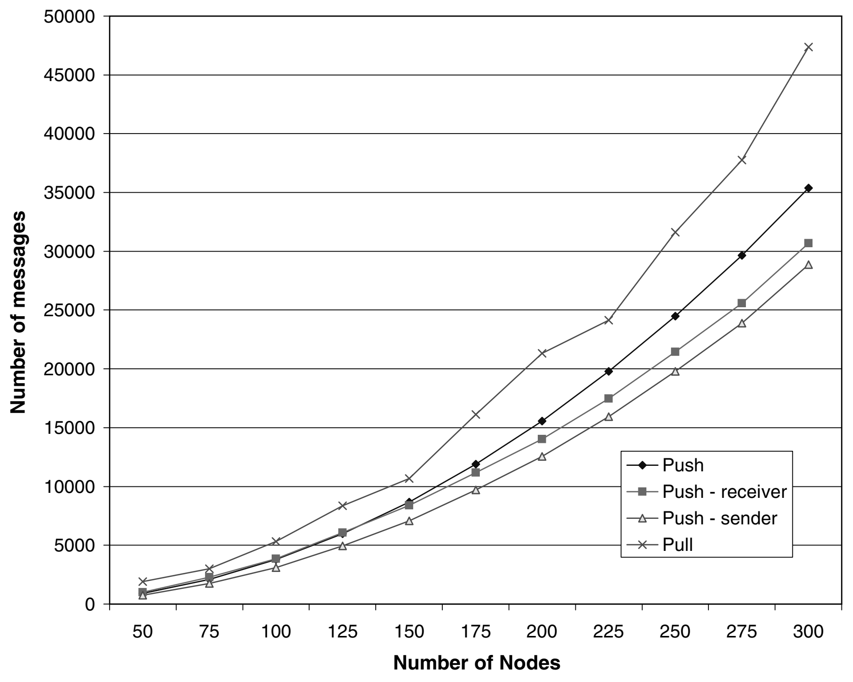


Fig. 6. The number of nodes varies and the percentage of connections is 20%.

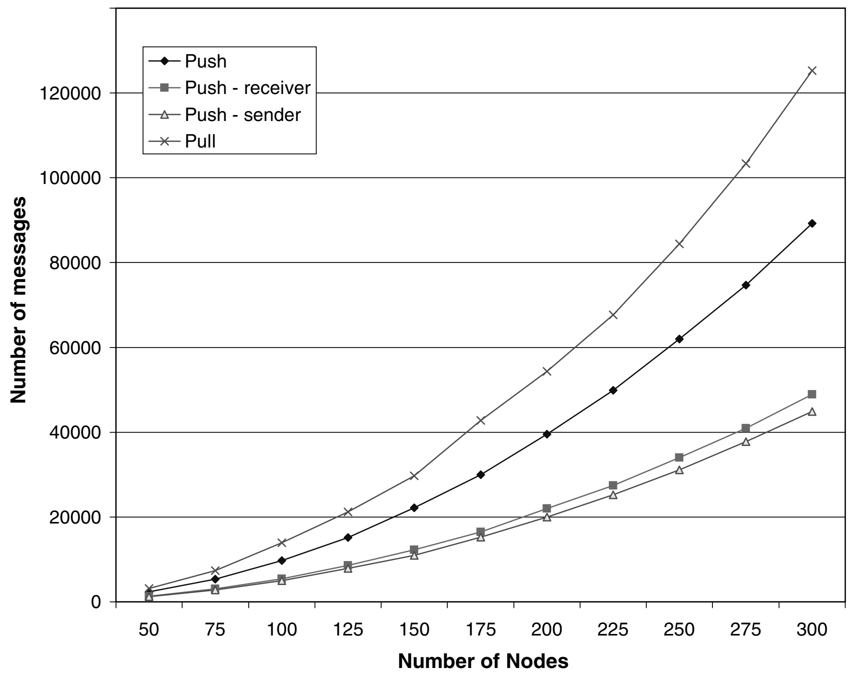


Fig. 7. The number of nodes varies and the percentage of connections is 50%.

messages that are used to propagate the update, with respect to the simple push technique presented in Section 2. Indeed the *sender-based* approach performs even better than the *receiver-based* one because there is no need to send a *cancel* message in the former. We also simulated a *pull-based* approach: every node polls the nodes with which they can communicate, with a frequency of 30 ms, and receives the update back if one of these nodes has it. To make the simulation real for this approach the simulator stops when all nodes have the update, but a node does not stop polling just after receiving the update, since the polling is continuously performed in such systems. We noted that a pull-based approach actually increases the number of messages. Moreover, by comparing Figs. 6 and 7, we also note that the difference between the normal push approach and the optimized ones increases as the percentage of connections increases: the number of messages tends to grow linearly.

This is generalized by the results in Fig. 8, where the number of nodes is fixed to 200, and the percentage is variable. This graph shows that when the percentage of connections goes over 50% the number of messages, generated by our algorithms, even

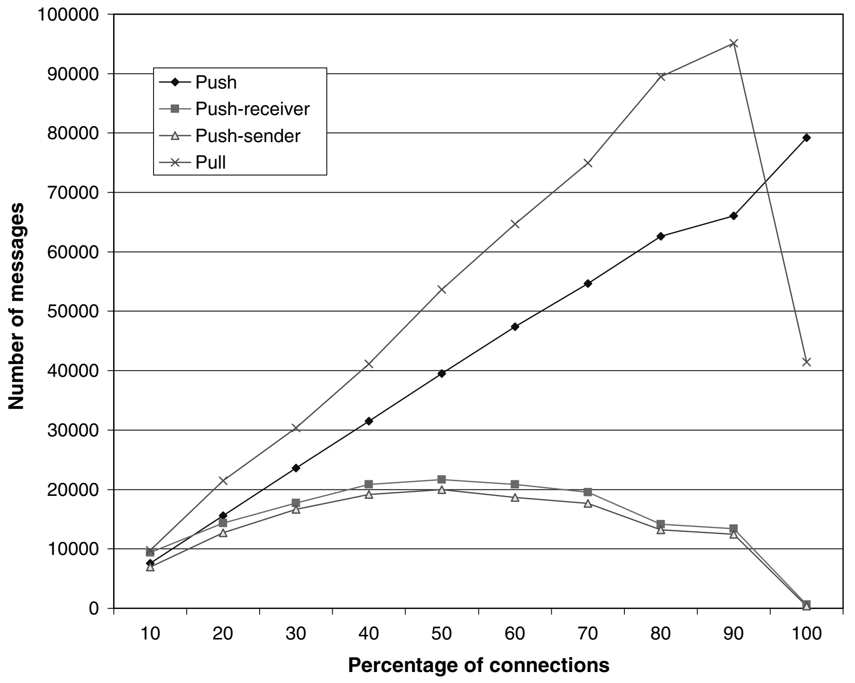


Fig. 8. The number of nodes is fixed to 200 and the percentage of connections varies.

decreases, and it gets very low when the graph is fully connected (100%). In this case, in fact, the issuer of the update sends the update to every one else, waits for the acknowledgment, and then, in the receiver-based approach, sends the cancel message; so actually no node gets the same update more than once (while this could happen in a non-complete topology). Note also that for a complete graph, the pull-based approach performs better than the normal push: indeed, as all nodes are connected to everyone, as soon as one has the update, the other ones will know about it at the first poll.

We also simulated the expiration of some time-outs for graphs with 100 nodes and 50% and 100% of connectivity (Figs. 9 and 10, respectively). The probability of time-out is variable from 0 to 1. The horizontal line in these graphs represent the normal push approach. We note that, unless the graph is fully connected, the receiver-based approach does not reach the number of messages produced by the normal push, even when the probability is 1. This is because the timed buffers give time to the system to get a better estimate of the other nodes (due to duplicate updates); so when the time-out expires, the node may already know that the ones, to which it is going to propagate the update, already have that update, and in that case many messages are saved. This is also a consequence of the fact that a time-out does not stop a node to propagate the update to other nodes which are not connected to the original sender. This does not happen in complete graphs where a time-out stops from propagating the update to anyone.

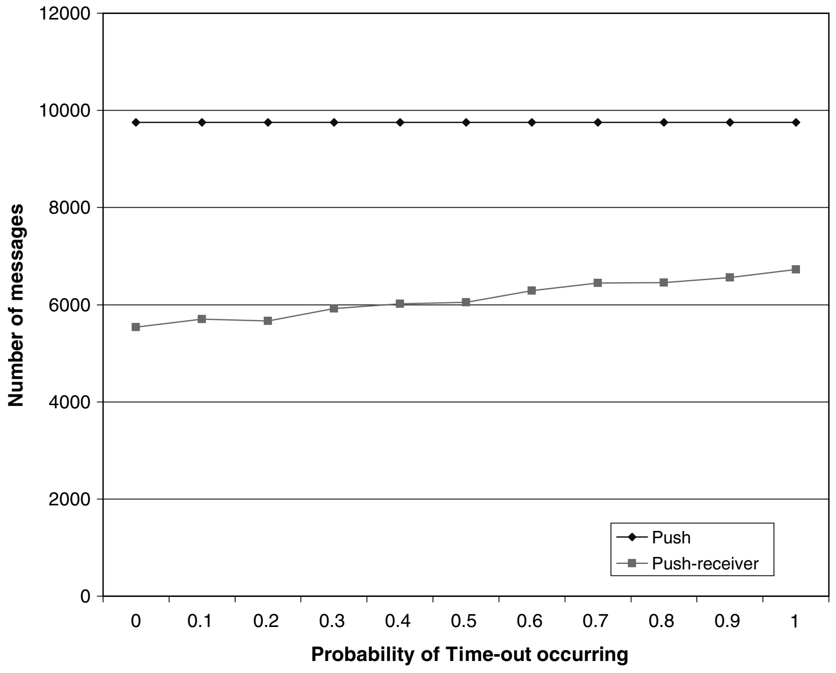


Fig. 9. The probability that a time-out expires varies (number of nodes, 100; percentage of connections, 50%).

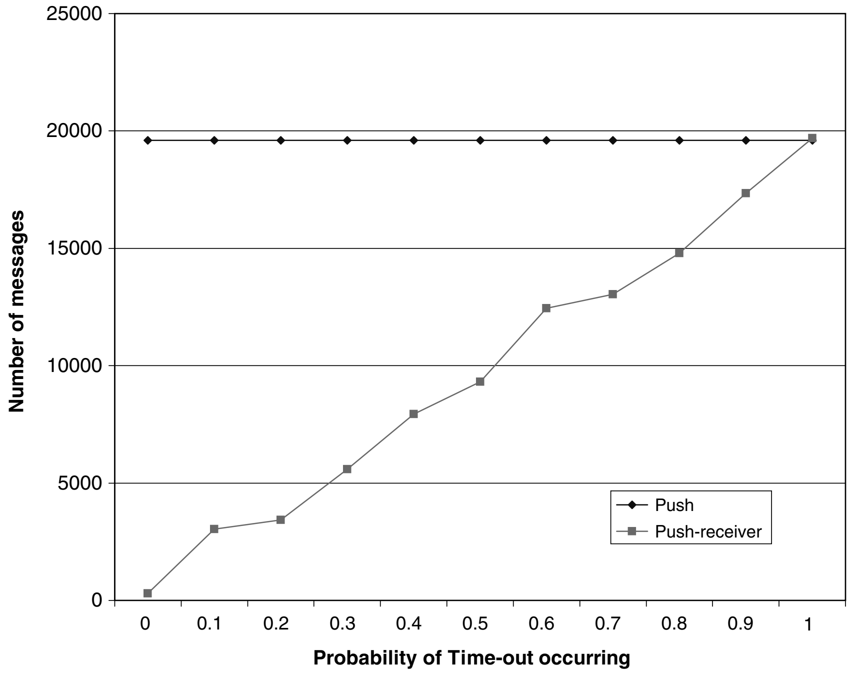


Fig. 10. The probability that a time-out expires varies (number of nodes, 100; percentage of connections, 100%).

In such a situation, when the time-out expires, the update is actually propagated to everyone, and indeed the algorithms behaves just like the normal push approach when time-outs expire with probability 1 (Fig. 10).

In Table 1, we show some simulation results on a slightly different kind of topology: we assume that 100 nodes are split into two sets, say *A* and *B*. Nodes in *A* are not connected to each other, while the nodes in *B* are connected to each other with a percentage of 80%. We consider that nodes in *A* are connected to some nodes in *B*, with a percentage of 2%, 5% and 10%, for each simulation; this means, for instance, that every node in *A* is connected to 2% of nodes in *B*. The numbers in the first column (5 and 20) are the numbers of nodes in the set *A*. The intent here is to simulate a net, where some nodes are static, and highly connected to each other (the nodes in *B*), while others are mobile and they communicate only with the nodes in this subnet, but not to each other (the nodes in *A*).

In this scenario it is interesting to note that in the case of (5, 5%), i.e., the set *A* contains five nodes connected to the nodes of the set *B* with a percentage of 5%, both for Push-receiver and Push-sender the number of messages is larger than (5, 2%) and (5, 10%). Actually the nodes in *A* can be seen as“drain” nodes since the messages are delivered to such nodes but a node in*A* will not propagate it to another node in *A*. In the scenario (5, 5%) the nodes in *B* do not have enough connections to the nodes in *A* to avoid sending duplicate messages; on the other hand, the number of these connections is large enough to waste messages. In the other two scenarios, the relations among the number of connections balance themselves to reduce the number of messages.

5.2. Time

We drew some results showing the time it takes for an update to be propagated to every node in the graph, using the several approaches presented in this paper. Every node stores the time they receive the update for the first time, and then the maximum for all the nodes is computed; this is the value that is shown in the graphical results. The time is always expressed in milliseconds.

In Fig. 11, we let the percentage of connections vary and compute such time. The three push-based propagations behave the same; this is consistent with one of our main goals: the optimized algorithms must not increase the latency for an update to be propagated. Indeed our two approaches let the receiver propagate the update immediately to the nodes that are not connected to the sender. The time for the pull approach is higher, as it depends on the frequency (in this graph it is set to 30 ms). When the graph is fully connected (100%) the time decreases since each replica can get the update from any other replica in the system.

Indeed the number of messages in a pull-based approach is independent of the frequency of polling (Fig. 12), but the time it takes for an update to be propagated in the whole system increases with that frequency (Fig. 13).

6. Conclusions and future work

In this paper we presented a new technique, *timed buffers* for propagating updates in a network where replicas can be mobile. Our goal is to provide all replicas with new updates as soon as

Table 1
Mixed topology

	Push			Push-receiver			Push-sender			Pull		
	2%	5%	10%	2%	5%	10%	2%	5%	10%	2%	5%	10%
5	15,306	15,370	15,398	1559	2551	1867	1436	2356	1752	22,662	22,348	23,005
20	10,802	10,958	11,298	842	1374	2796	758	1254	2606	21,904	16,346	16,086

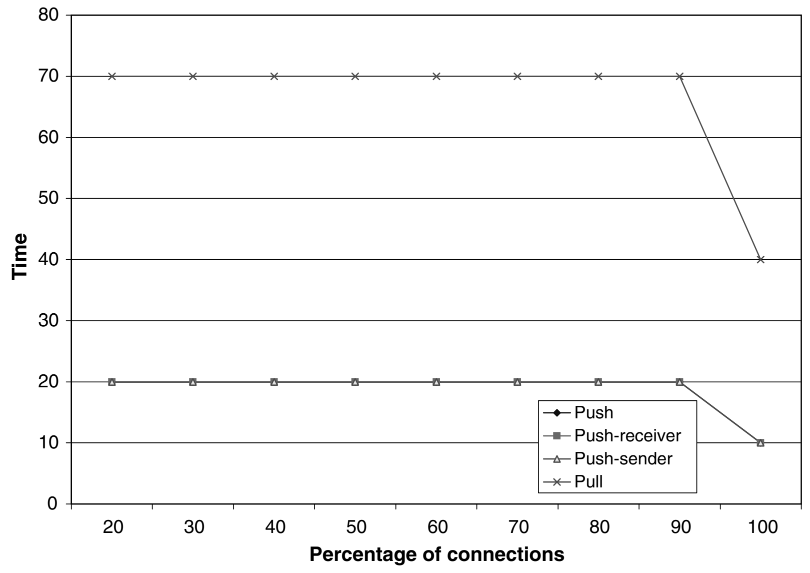


Fig. 11. Time for an update to be propagated (number of nodes: 200).

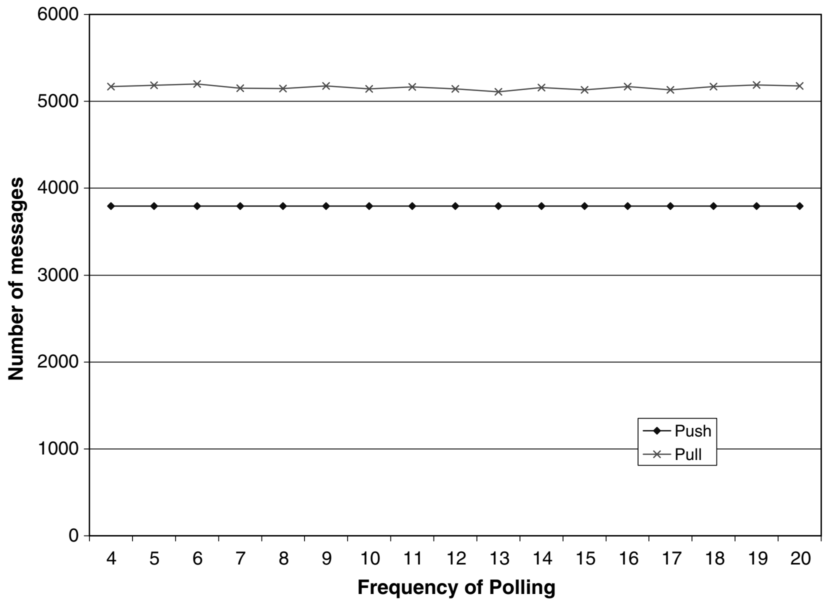


Fig. 12. The frequency of polling varies (number of nodes, 100; percentage of connections, 20%).

possible, while keeping the total number of delivered messages low. We use timestamp matrices both to stay conservative and to avoid sending updates to nodes that are known to be already up-to-date for sure. Vectors are not enough since we use a multi-master, push-based approach.

In [23], it is stated that with pull-based transfers, commitments tend to be executed more quickly than with push-based transfers. This makes sense if we consider that a push model may have to propagate more updates in order to stay safe; however, by using timestamp matrices to estimate the status of the other updates, and through the optimizations we describe at the end of this section, we believe that updates could be epidemically transferred faster, especially compared to a polling model (Section 5). Results in Fig. 9 also show that, if the topology of connections is not complete, our proposed push-based approaches react well to time-out expirations; this shows that time-outs give time to the system to achieve a more precise estimate of the other nodes' state.

Even if we do not address groupings explicitly we still believe that our system scales pretty well even in very highly mobile environments. Some problems with these implicit treating of the (possibly frequent) changing of network topology could arise when the number of replicas (especially if they are completely connected) enormously increases. However, in such cases, these systems would still be hardly tractable, or at least they would show a very low performance anyway.

Finally, our solution seems to be quite open to other optimizations; for instance we could reduce the size of the timestamp matrix which is piggybacked on every message that is sent and which may tend to grow in size if the number of replicas that share the same object grows. A possible optimization could be sending only the differences from the last matrix, and thus only *deltas* of timestamp matrices can be safely sent. This way very few information are sent and also the merging operation will be much faster, since most of the work has already been done.

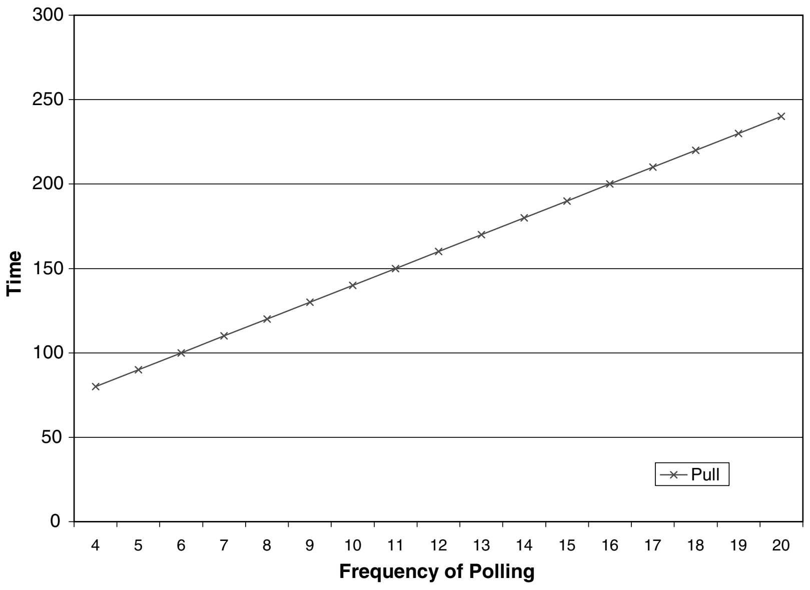


Fig. 13. Time for an update to be propagated (number of nodes, 100; percentage of connections, 20%).

Note that this does not prevent from sending useless information to a node (in fact its timestamp matrix could already be up-to-date), but at least information that would surely be superfluous is not sent.

In order to save memory space, a node can store in memory only the rows for the nodes it communicates to (the *active* nodes), and store the other ones (*inactive* nodes) in a backing store memory, for instance in a file; these rows can then be loaded on demand, when they are necessary, e.g., when a delta for them is received or upon reconnection with one of these nodes. Thus, the matrix will be divided in an *active set* and in a *non-active set*. This could be seen as an implicit implementation of grouping in domains inside the timestamp matrix [21]. Finally, we could reduce the size of messages that are sent, by using a mixed *push–pull*:

- Upon receiving (or creating) an update a node propagates a message with only deltas of its timestamp matrix (i.e., it communicates to the other nodes that it has some new updates).
- Upon receiving such a message a node checks if it's missing some of these updates, and in case it requests them to the sender.

This way we use the push-based approaches, described in Section 3, for propagating deltas of timestamp matrix, but without sending the update itself; the update will have to be explicitly requested by the other nodes (only once), so, in this case, a pull-based approach is used, but without polling and its drawbacks. This can be quite useful when the information to be transmitted are huge. In this case the push-based approach (with our proposed protocols) can be used to (conservatively) notify the replicas that more recent data are available, and the replicas in turn can decide whether to request the real data or not. Note that this can be adopted also if the entire new state of a shared object should be delivered instead of using a log-based approach.

Acknowledgments

I am very grateful to Ant Rowstron for his help and contributions on timed buffers. Mark Shapiro provided insightful comments about timed buffers.

The anonymous referees provided many helpful suggestions for improving the clarity and the style of the paper.

References

[1] D. Agrawal, A. El Abbadi, The generalized tree quorum protocol: an efficient approach for managing replicated data, ACM Transactions on Database Systems 17 (4) (1992) 689–717.
[2] S. Albers, New results on web caching with request reordering, in: SPAA: Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2004, pp. 84–92.
[3] N.T.J. Bailey, The Mathematical Theory of Infectious Diseases and its Applications, second ed., Griffin, 1975.
[4] S. Balasubramaniam, B.C. Pierce, What is a file synchronizer? in: Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MOBICOM-98), ACM Press, 1998, pp. 98–108.
[5] P.A. Bernstein, E. Newcomer, Principles of Transaction Processing, Morgan Kaufmann, San Francisco, 1997.
[6] K.P. Birman, T.A. Joseph, Reliable communication in the presence of failures, ACM Transactions on Computer Systems 5 (1) (1987) 47–76. Also Technical Report TR85-694, Cornell University, Computer Science Department.
[7] L. Cardelli, Global computation, in: ACM Computing Surveys, 28(4es), 1996, Article 163.
[8] U. Çetintemel, P. Keleher, M. Franklin, Support for speculative update propagation and mobility in Deno, in: Proceedings of the 21th International Conference on Distributed Computing Systems (21th ICDCS'2001), IEEE, 2001.
[9] U. Çetintemel, P.J. Keleher, Light-weight currency management mechanisms in mobile and weakly-connected environments, Journal of Distributed and Parallel Databases 11 (1) (2002) 53–71.
[10] U. Çetintemel, P.J. Keleher, B. Bhattacharjee, M.J. Franklin, Deno: a decentralized, peer-to-peer object-replication system for weakly connected environments, IEEE Transactions on Computers 52 (7) (2003) 943–959.
[11] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic algorithms for replicated database maintenance, ACM Operating Systems Review, SIGOPS 22 (1) (1988) 8–32.
[12] W.K. Ewards, E.D. Mynatt, K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, Designing and implementing asynchronous collaborative applications with Bayou, in: Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-97), ACM Press, New York, October 14–17, 1997, pp. 119–128.
[13] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, S. Wicker, An Empirical Study of Epidemic Algorithms in Large Scale Multihop Wireless Networks, Technical Report IRB-TR-02-003, Intel Research, 2002.
[14] E. Grosse, Repository mirroring, ACM Transactions on Mathematical Software 21 (1) (1995) 89–97.
[15] I. Gupta, A.-M. Kermarrec, A.J. Ganesh, Efficient epidemic-style protocols for reliable and scalable multicast, in: Proceedings of the 21st Symposium on Reliable Distributed Systems (21st SRDS'02), IEEE Computer Society, 2002, pp. 180–189.
[16] M. Herlihy, A quorum-consensus replication method for abstract data types, ACM Transactions on Computer Systems 4 (1) (1986) 32–53.

[17] J. Holliday, D. Agrawal, R. Steinke, A.E. Abbadi, Epidemic algorithms in replicated databases, IEEE Transactions on Knowledge and Data Engineering 15 (5) (2003) 1218–1238.
[18] J. Howard, S. Katz, Reconciliations, in: Symposium on Principles of Distributed Computing (PODC'94), ACM Press, 1994, pp. 14–21.
[19] S. Jajodia, D. Mutchler, Dynamic voting, in: U. Dayal, I.L. Traiger, (Eds.), Proceedings of the ACM Special Interest Group on Management of Data 1987 Annual Conference, 1987, pp. 227–238.
[20] M. Jelasity, A. Montresor, Epidemic-style proactive aggregation in large overlay networks, in: Proceedings of the 24th International Conference on Distributed Computing Systems (24th ICDCS'2004), IEEE Computer Society, 2004, pp. 102–109.
[21] T. Johnson, K. Jeong, Hierarchical Matrix Timestamps For Scalable Update Propagation, Technical Report TR96-017, University of Florida, 1996.
[22] B. Kantor, P. Lapsley, Network News Transfer Protocol, RFC 977, U.C. San Diego, U.C. Berkeley, February 1986.
[23] P. Keleher, Decentralized replicated-object protocols, in: Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99), May, 1999.
[24] A.-M. Kermarrec, I. Kuz, M. van Steen, A.S. Tanenbaum, Towards Scalable Web Documents, Technical Report IR-452, Vrije Universiteit, October, 1998.
[25] A.-M. Kermarrec, A. Rowstron, M. Shapiro, P. Druschel, The IceCube approach to the reconciliation of divergent replicas, in: Principles of Distributed Computing (PODC), ACM, 2001, pp. 210–218.
[26] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM, July, 1978, pp. 558–565.
[27] F. Mattern, Virtual time and global states in distributed systems, in: Proceedings of the International Workshop on Parallel and Distributed Algorithms, Gers, France, North-Holland, 1988, pp. 215–226.
[28] D.L. Mills, Improved algorithms for synchronizing computer network clocks, in: Proceedings, 1994 SIGCOMM Conference, London, UK, 1994, pp. 317–327.
[29] P. Mockapetris, K. Dunlap, Development of the domain name system, in: Proceedings of the ACM SIGCOMM'88, 1988, pp. 11–21.
[30] C. Oppermann, Microsoft Windows 2000 Active Directory Programming, 2001.
[31] B.C. Pierce, J. Vouillon, What's in Unison?, A Formal Specification and Reference Implementation of a File Synchronizer, Technical Report MS-CIS-03-36, Department of Computer and Information Science, University of Pennsylvania, 2004.
[32] B. Pittel, On spreading a rumor, SIAM Journal on Applied Mathematics 47 (1) (1987) 213–223.
[33] D. Rasch, R.C. Burns, In-place Rsync: file synchronization for mobile and wireless devices, in: USENIX Annual Technical Conference, FREENIX Track, USENIX, 2003, pp. 91–100.
[34] D. Ratner, Roam: A Scalable Replication System for Mobile and Distributed Computing, Ph.D. thesis, University of California, Los Angeles, 1998. Technical Report UCLA-CSD-970044.
[35] D. Ratner, G. Popek, P. Reiher, The Ward model: a scalable replication architecture for mobility, in: Proceedings of the Workshop on Object Replication and Mobile Computing, 1996.
[36] D. Ratner, P. Reiher, P. Gerald, Dynamic Version Vector Maintenance, Technical Report 970022, University of California, Los Angeles, Computer Science Department, June 30, 1997.
[37] D. Ratner, P.L. Reiher, G.J. Popek, R.G. Guy, Peer replication with selective control, in: Proceedings of the Mobile Data Access, First International Conference (MDA'99), Lecture Notes in Computer Science, vol. 1748, Springer, 1999, pp. 169–181.
[38] P. Reiher, J. Popek, M. Gunter, J. Salomone, D. Ratner, Peer-to-peer reconciliation based replication for mobile computers, in: Proceedings of the ECOOP Workshop on Mobility and Replication, 1996.
[39] Y. Saito, B.N. Bershad, H.M. Levy, Manageability, availability, and performance in Porcupine: a highly scalable, cluster-based mail service, ACM Transactions on Computer Systems 18 (3) (2000) 298–332.
[40] Y. Saito, H. Levy, Optimistic replication for Internet data services, in: Proceedings of the 14th International Conference on Distributed Computing, Spain, 2000.
[41] Y. Saito, M. Shapiro, Optimistic replication, ACM Computing Surveys 37 (1) (2005) 42–81.
[42] M. Shapiro, A. Rowstron, A. Kermarrec, Application independent reconciliation for nomadic applications, in: Proceedings of the SIGOPS European Workshop: Beyond the PC: New Challenges for the Operating System, Kolding, Denmark, 2000.
[43] B. Silaghi, Replication Techniques for Peer-to-Peer Networks, Ph.D. thesis, Department of Computer Science, University of Maryland, 2003.
[44] R. Thomas, A majority consensus approach to concurrency control for multiple copy databases, ACM Transactions on Database Systems 4 (2) (1979) 180–209.
[45] G. Voekler, On the scale and performance of cooperative web proxy caching, in: SC2000: High Performance Networking and Computing, ACM Press and IEEE Computer Society Press, 2000, pp. 41–42.
[46] D. Wessels, K. Claffy, RFC 2187: Application of Internet Cache Protocol (ICP), version 2, September, 1997.
[47] G.T.J. Wuu, A.J. Bernstein, Efficient solutions to the replicated log and dictionary problems, in: Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, BC, Canada, 27–29 August, 1984, pp. 233–242.