Enhancing the Availability of Networked Database Services by Replication and Consistency Maintenance

Hendrik Decker, Francesc Muñoz, Jordi Bataller, Luis Irún, Pablo Galdámez and Josep Bernabéu

Institut Tecnològic d’Informàtica
Universitat Politècnica de València
46071 València, SPAIN
{hendrik,fmunyz,bataller,lirun,pgaldam,josep}@iti.upv.es

Abstract We describe an operational middleware platform for maintaining the consistency of replicated data objects, called COPla (Common Object Platform). It supports both eager and lazy update propagation for replicated data in networked relational databases. The purpose of replication is to enhance the availability of data objects and services in distributed database networks. Orthogonal to recovery strategies of backed-up snapshots, logs and other measures to alleviate database downtimes, COPla caters for high availability during downtimes of parts of the network by supporting a range of different consistency modes for distributed replications of critical data objects.

1 Introduction

Database-supported service availability is strategically critical for an increasing number of business processes. Traditionally, ensuring availability has meant to take measures for avoiding negative ramifications of database downtimes. That is, downtimes are tentatively absorbed by high investments in additional hardware and attempts to marginalize downtimes by sophisticated strategies of backups and recovery.

With the advent of the internet and its role as a backbone of networked databases, an orthogonal way of enhancing the availability of data has become feasible, viz. by replicating data objects in several network nodes. However, the drawback of replication is a potential overhead for maintaining the consistency of replicated data.

In this paper, we propose a solution to reconcile the conflicting goals of enhancing the availability of services by replication of data, on one hand, and maintaining the consistency of replicated data, on the other. We describe the COPla platform, its architecture, its support for database replication and two of the consistency protocols that have been implemented on it. Embeddings of other consistency protocols in COPla have been described in [10]. Consistency protocols as used in COPla have also been described in [11] [13], however without details about their interaction with COPla. Devoid of a closer look at consistency protocols, an overview of COPla in general has been given in [12].

* This work has been partially supported by the EU grant IST-20997.
In a network of distributed databases, service applications may start multiple database sessions. Each service, or even each session, may use different consistency modes, according to the particular needs of a given service. Such networks may be used by enterprises that have several branch offices, such as banks, hypermarkets, etc. While some network services may use on-site generated data in local branches, others may also use information generated in other branches and offices. However, the availability, timeliness and reliability of answers to queried data may be severely hampered and network traffic may be exasperatingly congested, since transmission channels, bandwidth, powerful hardware etc may be a scarce resource. Particularly for web services in wide area and mobile networks of databases, the web may be very busy, and services may have to cope with network partitions caused by downtimes of local nodes, broken links to networked neighbors and the like.

To avoid such downsides and downtimes of web services, networked databases supporting these services are in need of effective means to warrant a sufficient degree of availability. This paper presents a platform which has been conceived to enhance the availability of networked database services. Somehow analogous to a flexible routing of data transmissions through networks where not all nodes are always up and running, the basic idea to ensure high availability of database services is to replicate data across the network while maintaining a sufficient level of their consistency.

Maintaining consistency means that, each time a data object is modified in a transaction, updates have to be multicast to a given number of database replicas (not necessarily to all, depending on the replication technique being used). So, database replication in performance-critical systems may seem to be advisable only if a sufficiently large share of accesses do not modify the data. However, an advantageous side effect of replicating data on distributed sites is that also “remote” accesses can be accomplished locally, thereby further improving accessibility and availability. After all, in a WAN, network partitioning by broken nodes or links are not at all uncommon.

One of the problems solved by COPla is to ensure consistency for transactions involving different database replicas. Solutions vary, depending on the kind of update propagation used: eager or lazy. COPla supports choices between different approaches to update propagation, each one being managed by a dedicated consistency protocol. Each such protocol is implemented by a particular COPla component. COPla’s API allows to deliberately switch from one consistency protocol to another, within a range of different ones, as needed.

Moreover, each consistency protocol supported by COPla may be run in any of three different consistency modes. Each session can change dynamically its consistency mode, thereby modifying the conflict handling rules for accesses to data objects that possibly are concurrently accessed by other sessions, in the same or other database replicas. So, COPla provides a rich toolset from which appropriate consistency guarantees can be chosen for satisfying the needs of the given service.

The first section below describes the structure and the functionality of COPla. Section 3 describes the COPla consistency management principles and outlines two of the consistency protocols implemented on the platform, an “eager” and a “lazy” one. Section 4 compares the eager protocol with other systems. Finally, section 5 summarizes the paper.
2 The COPla Architecture

The COPla architecture is structured by three layers, which interact via CORBA interfaces, thus enabling the placement of different layers in different machines. Hence, a multitude of simultaneous services (running on different nodes) is enabled to access the same database replica. All updates applied to these replica are propagated to other database replicas using COPla’s replication management components. The three layers, as depicted in figure 1, from bottom to top, are:

- **Uniform Data Store (UDS).** This component manages all persistent data. It directly interacts with a relational DBMS, storing there the persistent objects of the given service and the metadata of the consistency protocol. It isolates the upper layers from the actual storage system used. In practice, support for different RDBMSs will be provided in the final release of the UDS (currently, it only manages PostgreSQL repositories). The schema definition of the databases is done using GODL, a simplified version of the ODMG ODL language [3].

- **COPla Manager.** The COPla manager is the core component of the COPla architecture. It manages database sessions (which may include multiple sequential transactions, working in different consistency modes) and controls the set of database replicas comprised by the given network. The COPla manager also provides some caches to improve the efficiency of database accesses.

  In this layer, a local consistency manager is included. It can handle several different consistency protocols. All conceivable consistency protocols share some common characteristics. For instance, each of them is bound to be rather optimistic, since no locks are used “a priori” to access objects. Hence, the consistency checks must be done at commit time. When a session commits, its updates are multicast by its local consistency component to all consistency components placed in other nodes.
The way this is done depends on the particular consistency protocol being used. In general, all of the communication between the networked databases is controlled by the local consistency manager.

- **COPla Programmer Library.** This library is the layer used by application services to access system services. It also provides some cache support and multithreading optimizations that improve the overall system performance. Services need not be installed in the same node where the COPla manager or the UDS are placed. They only need to have the library layer on their nodes.

Between each pair of consecutive layers, there is a CORBA interface. So, each layer can be placed in a different node, since communication across layers is enabled by CORBA. The current system release is implemented in Java, and the Java ORB of the Sun J2SDK is used.

## 3 The COPla Consistency Management

In this section, we outline two (of several) consistency protocols that have been implemented on the COPla platform (others are described in [10]). The first one is an eager, the second a lazy protocol. The presentation of these protocols, in subsections 3.3 and 3.4, is preceded by subsections 3.1 and 3.2, in which general principles of the COPla consistency management approach are described, which are common to all consistency protocols. Subsection 3.5, about the principles of fault tolerance as observed in the COPla consistency management approach, is following up on the descriptions of the protocols themselves. Although these principles are common to all protocols implemented on COPla, we have not included fault tolerance in 3.1 because a more detailed look into the protocols themselves is conductive to a better understanding of 3.5. More details about failure handling, role migration between nodes and node recovery are covered in [11] [13]. Finally, subsection 3.3 is supplemented by subsection 3.6, on experimental performance measurements that have been done for the eager protocol.

### 3.1 Roles of Network Nodes

Considering a given session that tries to commit, the nodes involved in its execution may have two different roles:

- **Active node.** The node where the COPla Manager that has directly served the session’s execution is placed.
- **Synchronous nodes.** All other nodes that have a COPla Manager. In these nodes, the session updates will be eventually received, if such updates exist. Note that read-only sessions do not generate any database updates. Hence, these sessions do not have any synchronous node.

Moreover, in a given session, multiple objects may have been accessed. Before committing a session, some checks have to be done to ensure that the accessed objects’ states were up-to-date. One of the nodes receives a distinguished role in these checks, and the others will accept its decisions.
Consequently, for each object, there exists its owner node. That is the node where the object was created; it is the manager for the access confirmation requests sent by the active nodes at commit time. The management of these access confirmation requests is similar to lock management, but at commit time. To this end, the owner node compares two object versions, the one sent in the request (which is the object version accessed by the requesting session), and the latest object version that exists in the database. If they are not equal, the request is denied and the session will be aborted because it has accessed an outdated object version. On the other hand, if they are equal and there is no other granted request in a conflicting mode (a conflict exists if one of the requests comes from a session that has modified the object), a positive reply is sent to that active node. An active node can commit a session if all access confirmation requests that it has sent have been replied positively.

### 3.2 Consistency Modes

A session can be considered as a sequence of “transactions” done in the same database connection. Each of this “transactions” can be done in one of the following consistency modes:

- **Plain consistency.** This mode does not allow any write access on objects. It guarantees that all read accesses in this mode follow a causal order. On the other hand, this mode imposes no restriction on the currentness of the objects being read. Thus, they may be outdated.

- **Checkout consistency.** This mode is similar to the traditional sequential consistency, although it does not guarantee isolation. Thus, if several sessions have read a given object, one of these sessions is allowed to promote its access mode to “writing”. However, if two of these sessions have promoted their access modes from reading to writing, one of them will be aborted.

- **Transaction consistency.** In this mode, the usual transaction guarantees: atomicity, sequential consistency, isolation and durability, are enforced.

A session always starts in plain mode. If the guarantees provided in this mode are not sufficient for the given service, it can promote its consistency mode to checkout or transaction. In these two modes, all accesses are temporarily stored until an explicit call to the commit() or rollback() operations is done (with the usual meaning of such operations). Once one of these operations have been done, the session returns automatically to plain mode. Thus, the programmer is able to choose the consistency mode of each session of a given service, and this consistency mode can be changed as needed while a session is running.

### 3.3 An Eager Protocol

In this subsection, we describe the FB consistency protocol (a historic name), which implements an eager update propagation strategy, by using “Full Broadcast” of session updates, once the session is allowed to commit. That is, the FB consistency protocol
broadcasts object updates to all synchronous nodes when a session is committing. Consistency conflicts among sessions are resolved using object versioning. To this end, the protocol uses some metadata tables in the database where the current object versions can be stored.

The FB protocol proceeds as follows:

1. In active nodes, sessions are created and executed without any additional check. They are allowed to proceed until they request their commit operation.

2. When a service application tries to commit one of its sessions, the COPla manager responsible for executing this operation informs the local consistency manager thereof, before applying the commit to the associated UDS. To this end, the COPla manager builds two sets containing the identifiers of all objects read and written in such a session. These are the session read-set and session write-set.

3. Once these sets have been received by the local consistency manager, the latter sends an access confirmation request to the owner of each of the objects in the sets. Such messages include the object identifiers, their accessed versions, the access modes (read or write) and the consistency mode used by the session (checkout or transaction, since plain mode does not need a commit operation).

4. The owner node of each object checks if this access confirmation request would conflict in any way with previous access confirmation requests granted to other sessions but not yet released. A conflict arises if the requesting session has written the object and there is another session that has previously obtained a write grant on the same object version.

   Additional conflicts depend on the consistency mode of the sessions involved in the check. If all sessions have used checkout mode, then a conflict only arises if the requesting session has modified the object and other read-access grants have been obtained previously by other sessions. But in checkout mode, a session does not run into conflicts by having read outdated object versions.

   On the other hand, if at least one of the currently committing sessions has used transaction mode, then conflicts arise when either the requesting session has used an outdated object version, or when there are multiple sessions accessing the object and at least one of the sessions has written it.

   If the owner finds that a conflict arises, then it answers the access confirmation request with a deny reply. Otherwise, it sends a grant reply and the session identifier is recorded as “granted” until it explicitly releases this grant in step 5 or 8.

5. When the active local consistency manager receives the replies, and if at least one reply denies the access confirmation requests, then the session is aborted. However, if all of them grant the request, then the session will commit.

   If the session has been aborted, then a release message is sent to the object owners that had replied using a “grant” message.

6. If, in the previous step, the session has been allowed to commit, then the consistency manager of the active node broadcasts the session updates to all database nodes that have a consistency manager. This is an atomic broadcast.
7. Once the update message is received, the active node for that session commits it. The synchronous nodes will also commit the session updates. But before doing so, they have to check that no local session has accessed any of the objects received in that update message. If such local sessions exist, they are aborted. Once the update has been completed, the consistency managers placed in the synchronous nodes check if they are the owners of some of the objects updated. If that is the case, the grants set in step 4 are released immediately. Since no explicit message is needed to do so, this accomplishes the protocol.

3.4 A Lazy Protocol

In this subsection, we describe a lazy consistency maintenance protocol which has been implemented on COPla, called the Session set update protocol. “Session set update” mean that, when updates are transferred to other nodes, not only the object changes are transmitted to their synchronous replicas, but all session updates (i.e., the session write-set) are transferred to each node which has a synchronous replica of at least one of the changed objects.

For supporting plain mode, an particular problem arises: before the updates of a session can be applied, all sessions preceding it in causal order need to have already been applied to the same database. Sometimes, however, this may have not yet been done. For instance, when an object has a deferred replica in a given node that has not received any update for a long period of time, and some of the objects with a synchronous replica in that node have been modified in the same session. The sequence of steps needed to cater for not yet applied sessions is as follows:

1. A request is sent to the object owner, asking for the session that has done the latest change on that object, and also for all preceding sessions according to causal commit order. Each node maintains a log of committed sessions until they have been applied to all system nodes. Thereupon, the session is removed from the log.

   The request also carries the local (and out-of-date) object version number for the requested object. So, the object owner is able to build the graph of precedent sessions, by tracing the SIDs stored in the object versions and scanning the logs.

2. The object owner replies by returning the mentioned graph. It includes the session that has caused the latest update on the requested object, and also all of its precedent sessions according causal commit order. This causal commit order is easy to find via the SIDs stored in the object versions. Since, the log also maintains the read- and the write-set of each session, the graph can be built inductively, as follows. The root is the session having caused the last update. For constructing any subsequent layer of the graph, only the read-sets of the current layer have to be inspected, and all sessions that appear in the object versions of those read-set objects are included in the next layer. When a session does not appear in the log, it can not be added to the graph, since this means that all its changes have been applied in each node.

   The graph built as described above only maintains the SIDs of the sessions, but neither their read- nor their write-sets.
3. The requester scans the received graph depth-first, starting at the leaves and removing from the graph all sessions the updates of which have already been applied to the local database. When the scanning process reaches a level where no session has been removed, it terminates. The resulting graph is returned to the node of the object’s owner.

4. The owner receives the returned graph and replies with the read- and write-sets of all sessions in the graph. These data are stored in the requesting node upon reception, thus terminating the retrieval of precedent sessions.

Now, we are in the position to go into the details of the Session set update protocol. In general, we assume reliability of message transport, in the sense of TCP/IP reliability, as well as the existence of a notification service which informs alive nodes about failures and recoveries of others.

As already said, this protocol transfers the whole set of session updates, each time a session commits. The set of updates is sent to each node which has a synchronous replica for at least one of the objects updated in that session. Consequently, plain mode can be easily supported, without further ado, i.e., plain-mode reads can be locally executed without any further message exchange. Each consistency manager executes the following steps:

- Every node maintains a log containing each session applied in its local database. A process for updating all nodes in the network is run asynchronously in each node. As soon as this asynchronous process has made sure that a session has been applied in each node, the session can be eliminated from the logs.

- When a node detects an out-of-date object \( N_o \) (say) in a read request, it locates the owner node of \( N_o \) and sends a request message to it in order to update its object copy. This request contains the identifier and version number of \( N_o \).

- The owner node receives the request and checks its meta-data for the set of causally dependent sessions, which are needed to update the requested object from the given version to the version held in the local database of the owner node. More precisely, the following is done.

  - The owner node checks the meta-data for the last session that has modified the requested object \( T_o \) (say). In that session, other objects may have been read. For convenience, let us denote the read-set of \( T_o \) by \( R(T_o) \).

  - For each object \( o_i \) in \( R(T_o) \), the node searches its log for each session \( T_j \) which has \( o_i \) in its write-set (i.e., each \( T_j \) which causally precedes \( T_o \)).

  - The node then constructs a graph representing the causal dependencies of \( T_o \) and each preceding \( T_j \). Starting from \( T_o \), this construction iterates inductively through layers of causal dependencies, in order to finally include all causal dependencies in the graph. The iteration ends when all the logged sessions with causal precedence have been included.

- The resulting graph (which, for each session, actually contains its SQL statements) is sent to the requesting node. The latter eliminates from the graph each session
which has already been applied to its local database. In general, a session $S$ can be eliminated from the graph when $S$ has been already applied in the requesting node. This is the case when each object in the write-set of $S$ has a lower version than the version in the local database.

It is important to note that a session $T$ in the graph cannot be applied to the local database when an object contained in its read-set has a higher version number than the one held in the local database (i.e., there exists a causally precedent session which is yet unknown to the requesting node). For being in the position to apply $T$, the out-of-date object must first be updated.

- When the session elimination process has terminated, the requesting node sends a message to the owner node, requesting the complete write-set (values and version numbers) of the "session" resulting from the compacted graph.
- The owner replies with a message containing the write-sets of each session included in the request. This information can be extracted from the meta-data tables, since all of these sessions have been locally applied by this owner node, which therefore knows about all of these write-sets.

“Plain” consistency mode (ensuring causal consistency preservation) is easily implemented. The following is needed to provide this functionality:

- For each node, a log of SID, read- and write-set of each applied session. (Note that this may imply a redundancy of logs.)
- A session remains logged until an asynchronous process has checked that the session’s updates have been applied to each node.

### 3.5 Fault Tolerance

Since data objects are replicated in several RDBMSs, COPla is able to tolerate failures of part of the system nodes. To this end, the following protocol details must be considered.

- **Session completion.** If the active node of a session fails before it completes the atomic broadcast of the session updates, the occurrence of such a session is unknown on the rest of nodes. Hence, the session has to be aborted when the active node recovers. However, if the update atomic broadcast has been completed, the session has been committed on all system nodes. In this case, the only node that probably has not committed that session is the active one. But this does not matter, since the session updates will be transferred to that node when it recovers, if needed.

Another undesirable effect of this kind of failure is related to the access grants that the session may have obtained. Since the session has not been committed, these grants would remain assigned to the faulty node, preventing other sessions from obtaining access to such objects. The solution to this problem is easy. When the access confirmation requests have been received by an owner node, the identifier of the node that has made the requests is memo’ed and associated to the requests
in some data structure of the owner. When the membership service notifies that a node has failed, this data structure is scanned and all grants assigned to it are automatically released.

- **Ownership role migration.** When a node fails, all objects that were created in there have lost their owner. To replace it, the node with the next identifier in increasing order is chosen as a temporary owner for such objects. This temporary owner retains its role until it also fails and is replaced by another one, or until the original owner recovers.

Moreover, some steps are needed to obtain the lists of access grants that the faulty owner had when it failed. COPla uses a membership service that notifies all live nodes about any membership change (either join or failure). When such a notification is received, each consistency manager scans its list of received access grants and builds a message containing all the information of all grants given by the faulty owner. This message is then sent to the temporary owner that will replace the faulty one. If a node does not hold any grant of this kind, it must send an empty message. The temporary owner collects all such messages and builds a list of its granted confirmation requests. New access confirmation requests are not replied until such a list has been rebuilt.

- **Node recovery.** The recovery steps needed when a node rejoins the network are thoroughly described in [7]. An outline of these steps is provided subsequently.

1. Once a given node has failed, all remaining live nodes memorize the OIDs of all objects updated in all sessions committed since then, and associate these notes to the faulty node identifier, i.e., they add the OIDs to a hash table or some data structure which is indexed by node identifiers. Hence, all live nodes have registered the same state.

2. Once a faulty node recovers and tries to join again the network, all alive nodes freeze their respective databases. To this end, if some session has started the commit protocol or has modified at least one of the objects owned by its local node, it is allowed to terminate. Other sessions are blocked until the joining node is integrated in the system.

3. Once the allowed sessions have terminated, the live nodes send a message to the joining one, communicating that their local databases are prepared for the joining procedure. The joining node waits for all such messages. When it has received all of them, it broadcasts a message to these nodes, indicating that it expects the database updates.

4. The aforementioned nodes of the network reply to this message by another one, including the contents of all objects whose OIDs can be found in the hash table described in the first step and that are owned by the replying node. Hence, the database updates are collected from different nodes.

In fact, this recovery protocol is used in all consistency protocols outlined in [10]. Although the distributed collection of database updates as described in the previous paragraph does not provide any advantage or inconvenience for the FB consistency protocol, it permits a fast collection when the updates are propagated lazily. Instead of using another approach for the FB case, we prefer to use the same recovery steps as for the lazy consistency protocols implemented on the COPla platform (cf. [11] [13]).
5. Later, the joining node applies all such updates using a newly created transaction, which will be committed when the whole set of replies has been received.
6. Finally, the joining node broadcasts another message, indicating that the joining process has terminated and thus allowing the blocked sessions to continue.

### 3.6 Performance Measurements

The COPla architecture allows for an object-oriented view of a given database (i.e., collection of data objects), which is physically mapped to the replicated DBMS where the data are actually stored. Replication and consistency maintenance entail some additional burden on performance.

We have taken some measurements of these extra costs, for the eager FB consistency protocol. In particular, we have considered two types of transactions, with only two objects. The first one just reads the data objects, whilst the second one reads both objects and updates one of them (at random). The tests have used networks composed by one, two and four nodes. A singular node has been assigned to be the owner of both objects in all tests. In all test cases, only one session has been created in each node, and all transactions of that session have been executed sequentially. Complete sequentialization of transactions is the worst possible case, since the multi-threaded support and the caches are not used in that case.

![Figure 2. Elapsed times in a network with only 1 node.](image)

Figure 2 shows the additional costs introduced by COPla. We compare the time required to execute a given number of transactions in a system consisting of a single
node. So, the consistency protocol is not used here, but all COPla layers are actually used to map the object accesses in the service layer to the accesses needed in the relational database. Hence, this scenario is not contrived, but serves best to measure the overhead introduced by the COPla approach. As will become apparent, the overhead is proportionally reduced when the number of nodes increases.

Four lines are shown. The lowest one corresponds to the times needed when read-only transactions use directly the JDBC support; i.e., without COPla. The other three correspond to a load of read-only transactions, another with 80% of read-only transactions and the last one with 50% of read-only transactions, all of them using the COPla services. The read-only COPla transactions have a cost 9 times higher than the JDBC read-only transactions. However, we have to consider that currently, JDBC does not provide any replication support nor an interface easily usable in object-oriented programming (at least when it is compared to the interfaces provided by an ODMG-compliant platform such as COPla).

Figure 3. Elapsed times in different network configurations.

Figure 3 compares the results obtained from different system configurations with a mix of 80% read-only and 20% update transactions. We consider five different configurations. The first one uses only one node. All the others use a system with two nodes. In the second and third configurations, only one of the two nodes executes all the transactions, so the updates have to be transmitted to the passive node. The results vary, depending on the ownership of the objects. If all transactions have been executed by the owner node, the differences with the one-node configuration are minimal. However, if
the transactions have been executed by the not-owner node, they require almost twice the time of the previous case, due to the access permission requesting and granting messages.

The last two configurations correspond to a two-node system where both nodes execute transactions. In this case, the load is balanced between them and there is not an appreciable difference between owner and not-owner nodes. Additionally, the overall time is lower than that of the one-node system.

We have taken other measurements with a 4-node network. When only one node executes the transactions and propagates the updates to the other ones, the measured times are equal to the 2-node network described above. This is the expected result, since the updates are broadcast, and its cost does not depend on the number of targets. On the other hand, when all the nodes directly execute transactions, the overall cost is reduced to approximately half of the time measured in the 2-node network, when both nodes have executed their sessions. This result is also reasonable, since the use of multiple nodes allows a better balancing of the system load.

4 Related Work

Current work in consistency protocols for replicated databases can be found using either eager [1,8,14] or lazy protocols [2,4,10]. Each has its pros and cons, as described in [5]. Eager protocols usually hamper the update performance and increase transaction response times but, on the positive side, they can yield serializable execution of multiple transactions without requiring too much effort. On the other hand, lazy protocols may answer read requests by stale data versions (or at least they require extra work to avoid that), but they improve transaction response times and allow disconnected operation.

COPla is a platform for both eager and lazy consistency protocols, but we focus on eager update propagation, in this section. In [14], a good classification of eager protocols is presented, according to three parameters: server architecture (primary copy vs. update everywhere), server interaction (constant vs. linear) and transaction termination (voting vs. non-voting). Among the eight alternatives resulting from combining the three parameters, only two of them seem to lead to a good balance of scalability and efficiency: those based on “update everywhere” and “constant interaction”. This is mainly due to the load distribution achievable with the “update everywhere” approach, i.e., a delegate server executes the transaction and broadcasts the changes everywhere. The election of such a delegate server is dynamic. Each transaction can choose a different delegate. Moreover, low communication costs result from a “constant interaction”, where the update broadcast is done just once, either at the beginning or end of the transaction, rather than for each transactional operation, as is the case in the “linear interaction” approach.

The FB protocol complies with these two parameters. It uses “update everywhere” (instead of “primary copy”), because each transaction is done initially at the node where it was initiated, independent of the accessed objects. It also uses “constant interaction”, since the updates are only broadcast at transaction termination, once the object version

\[1\] We are planning to include the abort rates in the final version of the paper, but currently we are unable to provide them, due to a bug in the protocol implementation.
checking has been done. Due to this version checking on the object owners’ nodes, the FB protocol must be classified as “voting termination”. Although “non-voting termination” approaches require less message rounds, they either need atomic reliable broadcasts (with total order delivery) if the updates are done at commit time, or all nodes need to execute completely all transactions, even those that finally will be aborted (if the broadcasts are done when the transactions start). Thus, at first sight, a “voting termination” approach seems better.

However, our design differs a bit from the guidelines provided in [14] for the “voting termination” approach. Control of the transaction termination is based in our case on object versioning. Hence, the votes consist only in checking the accessed object versions, verifying that they have been the latest ones. We do not need a total order broadcast nor a 2PC to find out if a transaction is allowed to commit or not. Indeed, in the best case, we only need a single round of requests and answers to do the voting, and this round does not use a total order. Thus, our solution requires lower costs than those referenced as examples in [14].

Finally, although our technique provides good results in terms of communication needs (i.e., delivery ordering and number of messages), the use of a communication tool that provides total order guarantees may simplify the recovery protocols when failures occur. Indeed, the recovery protocols described in [9] allow that the previously running nodes do not block when a faulty node recovers. This advantage currently is not available in our system.

5 Conclusions

Orthogonal to traditional approaches for enhancing database availability, the COPla architecture caters for the availability of data and services by supporting the consistency maintenance of replications over a multitude of network nodes. Within COPla, a range of different consistency protocols are provided. Depending on the needs of a given variety of application services, COPla users may choose from the set of available consistency protocols the one which fits best, in each particular case. Moreover, COPla supports three different consistency modes for each of its consistency protocols, which can be chosen at will for each transaction.

In this paper, we have described one of several consistency protocols that are available in the current version of our system. It is based on eager update propagation, but does not need a total order broadcast communication nor multiple update rounds. Hence, it minimizes the communication needs of such kind of protocols, thus reducing the usually long transaction completion time, which otherwise is one of the main drawbacks of eager protocols.

As this draft is on the verge of being submitted, improved evaluation results of performance measurements are being reported. They are mainly due to incremental improvements of the implementation of several COPla modules. They seem to have accumulated into a nice measurable upgrading of the overall network performance, both for primitive and for more complex scenarios, i.e., for networks with few or more nodes. We are expecting to include a comprehensive presentation of the new results in the next version of this paper.
References