GlobData: A Platform for Supporting Multiple Consistency Modes*

F. Muñoz, L. Irún, P. Galdámez, H. Decker, J. Bernabéu, J. Bataller, M.C. Bañuls
Instituto Tecnológico de Informática, Universidad Politécnica de Valencia, Spain
{fmunyoz,lirun,pgaldam,hendrik,josep,bataller,banuls}@iti.upv.es

ABSTRACT
GlobData is a platform that provides an object-oriented view of wide-area-networked relational databases with replicated data for ensuring high availability. We discuss the embedding of protocols in GlobData, for maintaining the consistency of replications. The protocols are able to alternate between three different modes of consistency. Modes can be changed on-line and per session, i.e., GlobData supports different and changeable consistency modes in simultaneous sessions.

KEY WORDS
Distributed Databases, Consistency, Availability

1. Introduction

In networked applications, data are often replicated for enhancing their availability. On the other hand, maintaining the consistency of replications frequently needs to lock data, thus conflicting with the requirement of high availability. Traditional approaches for replicating databases are based on fast LANs [1, 2, 3, 4], using network-intensive protocols. However, in applications based on WANs (e.g., the internet), the network is a limited resource, and the ensuing problems must be appropriately dealt with [5, 6].

GlobData [7] strives to solve the aforementioned problems, focusing on applications in which efficient recovery from failures, availability and high volumes of data processing are key requirements. It does so by defining a specific software architecture for networked data replications, together with an API and a choice of flexible consistency modes for data access.

In this paper, we propose protocols which meet the consistency and availability requirements posed by GlobData. Due to space limitations, we focus on their essential characteristics. More details are elaborated in [8]. Besides an optimistic support of lazy replication, the most notable feature of the protocols is their ability to alternate online between three different consistency modes. Transactions are allowed to proceed locally, being checked for consistency violations only at commit time. When a consistency violation is detected, the transaction is rolled back.

Moreover, the GlobData architecture allows the integration of different consistency protocols, if they use the appropriate interfaces. As a result, besides the protocols outlined in this paper, other protocols are being designed, implemented and integrated in this system [9].

The rest of the paper is structured as follows. Section 2 introduces basic concepts and characteristics of the GlobData consistency policy. Section 3 outlines the algorithm underlying each protocol. Section 4 describes more details of one of the protocols. Section 5 and 6 briefly discuss failure management and recovery. In section 7, we conclude.

2. Common Concepts

Within GlobData, the database nodes communicate via local consistency agents. They can be viewed as mediators for every data access and each transaction performed by the local sessions. Because of lazy replication, not all nodes necessarily have the latest version of each data object. Thus, for each data object, each database node assumes one of the following roles:

- **Owner node**: Initially, that role is assumed by the node where the data object has been created. It handles access confirmation requests for the object; i.e., it allows or denies such requests when asked by the initiator of a session.

- **Synchronous nodes**: These nodes maintain up-to-date replicas. For each object, the synchronous nodes and their number are supposed to be pre-configured.

- **Deferred nodes**: They usually do not store up-to-date replicas, although they may do so, sometimes (e.g., when one of them has initiated the session which has caused the latest change of the object).

Our approach replicates data lazily. To each object, an owner node is assigned, for controlling accesses to that object. However, control is leveraged only when a session wants to commit. Then, the accessed objects' owners are requested to check if the session has used the latest committed versions. If so, the owners validate the accesses, and the session can commit. If any of the read objects is outdated, the session is flagged for abortion. We call the requests for these checks "access confirmation requests", since the owners are asked to check only those objects that have been used in the given session.

To cope with failures of nodes and partitioning faults, a notification service is needed. It assigns static identifiers to each database node in the network. The

---

*This work has been partially supported by the EU grant IST-1999-20997 and the Spanish grant TIC99-0280-C02-01
protocols need these identifiers for recording the object ownership and also each node’s role.

Since we use access confirmation request management, each session identifier (SID) must include the identifier of the node that has initiated the session. Thus, the protocols know which granted access requests did belong to a faulty node, which is necessary to manage a node’s failure.

For each object, an object identifier (OID) identifies its initial owner. When a node fails, ownership is transferred to another node. Clearly, this change of ownership needs to be noticed by all alive nodes. Besides OIDs, also a version number is associated to each object, for the purpose of node recovery. A recovering database then only needs to send to one of its neighbors the last known version number of each of its objects. The receiving node replies with all changes to be applied, for re-establishing an up-to-date state. An object version number includes the SID of the session that has written its last value. The purpose of that is explained in sections 3.3 and 4.

When some network links break and only some mutually isolated subgroups of nodes remain connected, access to an object is granted only if the subgroup containing that object contains more than 50% of alive nodes with up-to-date replicas. The ownership roles of a faulty node are transplanted to one or several of the remaining alive nodes (if any) with replicas of the owned objects.

3. An Outline of the Protocols

Each of our protocols proceeds as follows: The consistency manager of each node maintains several metadata tables, containing information about the versions of locally stored objects.

When a session wants to read an object, the local consistency manager checks its version. If outdated, it sends a message to the owner, requesting the recent updates. The session is blocked until these updates have been applied. Only then, the read access can be executed. As opposed to that, write accesses are executed without any check.

For each access, the sets of accessed objects are entered into the meta-data tables, for building the read- and writeset. When a session wants to commit, the local consistency manager retrieves the read- and write-sets and sends an access confirmation request to the accessed objects’ owners including OID, version number and access mode (read or write) of each object.

When an owner receives one of the above messages, it compares the object’s version number with its latest version. If equal, the owner grants access and sets the SID to be the access grant’s tenant. The grant is exclusive if the access mode was “write”, and shared if it was “read”. The grant persists until the session has propagated its updates to the synchronous nodes with replicas of the accessed objects. Once that is done, the session initiator updates the version number of the write-accessed objects and then releases the access grants.

If a session holds an access grant, other sessions requesting the same grant in a conflicting mode receive a service denial. Then, the consistency manager which requested that grant aborts the corresponding session.

3.1 Different Consistency Modes

Different sessions may use different consistency modes, which however may change for each transaction. In increasing order of restrictiveness, the three modes are:

- **Plain**: It only allows isolated read requests and guarantees that all accessed objects respect a session-causal commit order. Thus, the accessed objects may not be up-to-date.

- **Checkout**: This is a permissive variation of the transaction mode (see below). Isolation is not guaranteed. Thus, if several sessions read the same object, only one of them is allowed to promote its mode to write mode. This may break serializability, thus blocking the promoting transaction in a standard transactional setting, or entailing abortion of the transaction.

- **Transaction**: The usual transaction properties of atomicity, consistency, isolation and durability must be provided.

In practice, when two sessions with different consistency modes conflict, priority is usually given to the more restrictive mode.

3.2 Protocol Alternatives

The basic protocol outline at the beginning of section 3 admits various alternatives of implementation. They engender different kinds of consistency protocols. One of them is described in the next section. The characteristics that admit alternative choices are:

- **Update multicast when a session commits**: A protocol may broadcast the session changes at commit time, in which case all replicas are synchronous. Thus, by properly observing the session commit order, all consistency modes can be easily guaranteed. Replication is lazy if the updates are multicast only to a preconfigured subset of replicas. Since different objects may reside in different sets of synchronous nodes, special care has to be taken for guaranteeing all consistency modes for lazy replication.

- **Per object or per session update propagation**: For lazy replication, all updates must either be propagated per session to each node with a synchronous replica of any updated object, or each update is propagated per object. Propagation per object implies: if two objects have been modified but not reside in the same set of synchronous nodes, then only the changes involving a single object are propagated to each set of synchronous nodes. Propagation by session entails that plain mode consistency is always satisfied (which otherwise is not guaranteed).
Depending on the consistency mode, the consistency manager takes the following steps when a session reads or commits.

- **On read accesses:**
  
  For plain mode, the object version in the local database need not be the latest one. But causal commit consistency must be guaranteed. So, a new session has to be applied to a database only when all sessions preceding the committing one have already been applied. So, plain mode is no problem for read accesses (though some care must be taken when commits using other modes were made).

  For read access in checkout and transaction modes, the local consistency manager has to ensure that the latest version exists, to avoid a rollback. If this version is not present, it has to be requested from the object's owner. The owner then returns the requested version to the session initiator. If plain mode is granted, not only the latest version is needed, but all versions between the one in the initiator's database and the latest one together with all other changes of sessions that have caused these updates.

- **On commit time:** There are no commits in plain consistency mode because only read access is allowed.

  In checkout mode, read accesses are treated as in transaction mode, but the commit procedure differs. If a session has read a version that is not the latest one at commit time, it is not aborted. However, objects in the write-set have to be checked at commit time: their version must be the latest one. If not, the session is aborted. Thus, read objects may change while the session is executing, but written objects must not be overwritten by other concurrently committing sessions.

  In transaction mode, both read and written objects must be up to date at commit time. If a subsequent change has been made by a concurrent session, the terminating session must be aborted.

### 3.3 Three Consistency Protocols

We present three consistency protocols with different characteristics, as follows:

- **Full object broadcasting:** This protocol implements immediate updates of all database replications, so it does not use lazy replication. Thus, the write-set of a committed session is broadcast and applied to all database nodes immediately. Of course, not all sessions are committed, since owners must grant the access confirmation to do so. These access permissions depend on the consistency mode used. The protocol supports each of the three consistency modes.

  **Simple object update:** This protocol uses lazy replication and object updates, instead of session updates. Although this protocol complies with all consistency modes, it requires more effort in plain mode, since the way updates are propagated in sessions using transaction or checkout mode does not entail the guarantees required for plain mode accesses.

  Note that, at commit time, the updates are only propagated to the preconfigured synchronous replicas of each modified object. Thus, the full effects of a session may possibly be not reflected at each node which has received an update message. That is, a node may have a synchronous replica of one of the objects involved, and a deferred replica of one of the other objects.

  When a read operation needs a more recent version than the one stored in the local database, only the latest version is requested (and obtained) from the owner. No other contents need to be transferred.

  **Session set update:** This protocol uses lazy replication and session updates. That is, when the 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 additional problem appears: before the effects 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, this may happen 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 get a group of missing sessions is as follows:

  1. A request is sent to the owner, asking for the session that has made 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 owner is able to build a graph of precedent sessions, by tracing the SIDs of the object versions and scanning the logs.

  2. The owner replies by returning the mentioned graph. Since the log maintains SIDs of object versions as well as read- and write-set of each session, the graph can be built inductively. The root is the session having caused the last update. For 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 all its changes have been applied in each node. Note that the graph only maintains the SIDs of the sessions, but neither read- nor write-sets.

  3. The requester checks the received graph and scans it
depth-first, starting at the leaves and removing from the graph all sessions that have already been applied to the local database. When the scanning 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 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.

4. More Protocol Details

This section gives additional details for the session set update protocol. We assume TCP/IP reliability of message transport and the existence of a service which notifies alive nodes about failures and recoveries of others. Each time a session commits, the whole set of session updates is transferred to each node which has a synchronous replica for at least one of the updated objects. Thus, plain mode reads can be locally completed without any further message exchange. Each consistency manager proceeds as follows.

- Every node maintains a log containing each session applied in its local database. A process for updating all nodes is run asynchronously in each node. After that is done for 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 receives the request and checks its meta-data for the set of causally dependent sessions, which are needed to update the requested object to the version held in the owner’s local database. 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. Let \(R(T_o)\) denote the read-set of \(T_o\).

  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 of causal dependencies of \(T_o\) and each \(T_j\). It starts from \(T_o\) and iterates through layers of causal dependencies. The iteration ends when all the logged sessions with causal precedence have been included in the graph.

- The resulting graph is sent to the requesting node.

  The latter eliminates from the graph each session \(S\) that has already been applied to its local database, i.e., when each object in the write-set of \(S\) has a lower version than the one in the local database.

Note that a session \(T\) cannot be applied to the local database when an object in its read-set has a higher version than the one in the local database (i.e., there is a causally precedent session which is yet unknown to the requesting node). For applying \(T\), the out-of-date object must first be updated.

- When the session elimination process is completed, the requesting node notifies the owner by 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. That can be extracted from the meta-data tables, since these sessions have been locally applied by this owner, which therefore knows about these write-sets.

Note that “plain” consistency mode is easily implemented. It ensures that each update preserves causal consistency. In order to provide this functionality, a log of every applied session for each node is needed. (Note that this may imply a redundancy of logs.) A session is kept logged until an asynchronous process has checked that the session has been applied to each node.

5. Fault Tolerance

Several failure scenarios must be considered for ensuring that the protocol is fault-tolerant. Subsequently, we briefly discuss three such scenarios. First, we specify the handling of failures in the protocol. Second, we address the migration of ownership roles. Third, we deal with node recovery.

5.1 Failure Handling

When a node fails, the assumed notification service monitors this failure and informs all alive nodes about it. Partition failures are notified the same way, but then, the set of faulty nodes may be larger. Let us see what happens with a session initiated by a node which has become faulty. We distinguish the following cases according to the step at which the failure occurs:

- If the session has not yet surpassed the access confirmation granting step of the protocol, no record of that session exists in any of the alive nodes. So, that session can be discarded. When its host node recovers, it must abort that session.

- If the session fails once it has obtained the remote access grants, but before it has multicast any update, a similar situation arises. No record of the session updates can be found in any of the live nodes, so the session cannot be terminated in the remaining nodes. Thus, that session must be aborted when its host node recovers. However, the faulty node has obtained some access grants, which may prevent other sessions from going ahead.
Another problem arises in the following situation.

Moreover, if an object update has been received by one of its synchronous replicas, all of them have received this update, because the multicasts are reliable. To deal with these potential situations, we propose the following. When the notification service communicates a node’s failure, all object owners scan their grants lists. If access to some object has been granted to a session initiated in the faulty node, the access grantor (i.e., the owner of the object) has to check if some update multicast associated to the SID which requested that access has been received. If no such update was received, the grant can be released. Otherwise, the following point has to be considered.

- If the session has at least initiated the update multicast, its updates may have arrived to other nodes. Then, the same as in the previous case must be done. The grants held by this session have to be released. Since the updates have been received in the original node (i.e., all grants have been obtained and all changes, though already applied, are still not committed in the original node), no additional access grant is needed. Since the session initiator’s node has failed, the grants are not needed by the node which replaced the faulty one, because the replacing node has already committed this session. Consequently, the access grants have already been used correctly and thus must be released now. With that, the session is completed.

No other case needs consideration. Possibly, the session had not been completed yet, but this only means that it held some grants that have been released as a consequence of the steps explained above.

- Another problem arises in the following situation. Suppose the node has failed once the update multicast for an object was initiated, but before all deferred replicas holding the latest version number so far have been notified that this version has changed and that they therefore do not have an up-to-date version any more. Then, the node which “inherits” the object ownership does not have the correct information on the version number held by each of the deferred replicas.

When the new object owner assumes its role, it already knows that it maintains a synchronous replica. It may decide that one or several other nodes should be updated to also hold a synchronous replica. (This decision depends on the particular replication policy in a given application.) Moreover, it has to broadcast a message to each accessible node which holds a deferred replica of an object whose ownership it has “inherited”. This message contains the version number of the newly owned objects. The deferred nodes will reply to this message by indicating whether their current replica is actually out-of-date or not.

5.2 Role Migration

When a node fails, each object it owns has to be managed by one of the remaining live nodes. Thus, object ownership has to be migrated. Two issues have to be considered. Firstly, the criterion for electing the node replacing the faulty one. Secondly, the migration of the access-granting management.

- Since each node has an identifier, it suffices to choose the alive node holding a synchronous replica with the lowest identifier among all identifiers that are greater than that of the faulty node (or the lowest one, if the faulty node had the greatest identifier). Note that it makes sense to appoint a new owner only if more than one half of synchronous replicas are still alive.

If an even number of synchronous replicas is used, a particular criterion is needed for breaking the tie in case of network partitioning. For instance, the subgroup holding the node with lowest identifier among the previous set of synchronous replicas may be determined to contain the new owner.

We assume that the number of synchronous replicas is known in advance. In case of network partitions, it may arise that the owner of a given object remains alive, but too many of its synchronous replicas remain unavailable. In that case, the current owner must give up its role. Then, the following problem may arise. If the majority of synchronous replicas are in another subgroup after the network partition, one of them will take up the management role, according to the strategy proposed in the previous paragraph. However, if those replicas would fail, no other session at all will be able to use that object again.

- The node which “inherits” the object ownership has to ask each other node about their access grants; i.e., it has to know which of the grants it manages has an owner and who is that owner. To this end, each node holding a grant that was managed by the faulty node, sends an ownership message to the new owner. This message contains the OID of the object associated to the grant, the access mode, and the SID of the session that holds it. If a node has no grants, it sends an empty message.

A timeout for receiving all of these messages is set by the new owner. If some message has not been received timely, an explicit request is sent to that node.

6. Node Recovery

When a node recovers, the notification service informs all alive nodes about it, for reconfiguring the system state. That is, the recovering node resumes ownership for each object it had initially created. Recall that one of the alive nodes had immediately “inherited” the object ownership for all objects of the recovering node. Upon recovery of the original owner, the intermediate owner
must send all information concerning access grants to the recovering node. As long as this message is not received by the recovered node, no incoming access grant requests or releases can be processed. Rather, they have to be buffered, and once the message and the object ownership have been transferred to the recovered node, the later resumes the access grant management.

For some object, an access request or release still may reach its intermediate owner, right after the latter has passed back the ownership to the original owner. In that case, each such message must be forwarded to the original owner. The sender of the message does not need to take care of this; forwarding is done by the intermediate owner, who knows the current (original) owner node.

Special attention is needed if the recovered process was initiated by a node which must maintain synchronous replicas of several objects. Each owner of such an object has to re-include it in the set of synchronous replicas, and possibly one of those replicas has to be degraded to the deferred category (i.e., it will eventually become obsolete if it is degraded). No extra message exchange is needed to do so.

Note that, during reconfiguration, no new sessions are allowed to begin their commit procedure, and the session management is temporarily disabled. Further note that the recovered node needs to receive all relevant updates having taken place during its down-time. To this end, each of the previously active nodes sends to this recovered node the latest state of the objects that have been modified while that node was faulty.

7. Conclusion

Global data access is increasingly important to a growing number of networked applications. The lockings necessary for maintaining replication consistency often obstruct the availability of data objects. In this paper, we have proposed protocols to reconcile availability and consistency, by a flexible choice among different consistency modes of a protocol implementing lazy replication.

The protocols are appropriate for applications which predominantly work with “local” data, of which replica are needed to ensure high availability. These protocols are currently being implemented within the GlobData project [7, 10]. Performance measurements will be presented and discussed in a follow-up paper.

References


